NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

TOWARDS AN IMPLEMENTATION OF POLYMORPHIC C

by

Peter Bryant Bonem

September 1995

Thesis Advisor: Dennis Volpano

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1995	Master's Thesis

4. TITLE AND SUBTITLE
TOWARDS AN IMPLEMENTATION OF POLYMORPHIC C

5. FUNDING NUMBERS

6. AUTHOR(S)
Bonem, Peter B

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Functional programming languages incorporate a number of powerful feaures, including advanced polymorphic type systems and first-class, higher-order functions. However, these important features have had little effect on popular imperative languages such as C. As part of the Advanced Type Systems Project at NPS, a dialect of C called Polymorphic C has been designed which integrates an advanced polymorphic type system into C.

In order to implement full parametric polymorphism while retaining the run time efficiency of C, it is necessary to allow mixed data representations. We recommend adopting a variant of the program translation methods first proposed by Leroy to implement mixed data representations in ML for use in Polymorphic C.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Polymorphism, Polymorphic Type Systems, Functional Programing Languages	104
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

TOWARDS AN IMPLEMENTATION OF POLYMORPHIC C

Peter B. Bonem
Lieutenant Commander, United States Navy
B.S., Purdue University, 1979

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1995

Author: Peter B. Bonem
Peter B. Bonem

Approved by: Dennis Volpano
Dennis Volpano, Thesis Advisor

Thomas Wu, Second Reader

Ted Lewis, Chairman
Department of Computer Science

ABSTRACT

Functional programming languages incorporate a number of powerful features, including advanced polymorphic type systems and first-class, higher-order functions. However, these important features have had little impact on popular imperative languages such as C. As part of the Advanced Type Systems Project at NPS, a dialect of C called Polymorphic C has been designed that integrates an advanced polymorphic type system into C.

In order to implement full parametric polymorphism while retaining the run time efficiency of C, it is necessary to allow mixed data representations. We recommend adopting a variant of the program translation methods first proposed by Leroy to implement mixed data representations in ML for use in Polymorphic C.

TABLE OF CONTENTS

ACKNOWLEDGMENT

I wish to thank Prof. Dennis Volpano for stirring my interest in programming languages and for his guidance in the preparation of this thesis. I am richer for having been associated with him in this small way. I regret that the time has been so short.

I also wish to thank Prof. Thomas Wu. His many insightful comments greatly improved this work.

I save special thanks for my lovely bride, Amy, whose love and encouragement are integral to everything I attempt. This work is as much hers as it is mine.

I. INTRODUCTION

Chapter I is divided into five sections. Section A introduces the functional programming paradigm and discusses the directions and some obstacles faced by researchers in the field. Section B is dedicated to concepts and definitions necessary to the entire scope of the thesis. Section C discusses some software engineering issues motivating research into polymorphic expression. Section D contrasts C and C++ with functional languages and discusses several ways in which C/C++ can be improved. Section E concludes the chapter with a brief introduction to Polymorphic C, a polymorphic imperative language being developed as part of the Advanced Type Systems Project at the Naval Postgraduate School and Florida International University [SmVo95].

A. FUNCTIONAL PROGRAMMING LANGUAGES

Much work has been done in the past decade on the development of advanced programming languages, particularly functional languages such as SML/NJ, Haskell, Miranda and Napier88. Functional languages all share the common paradigm that function application is the primary method used for computation. This paradigm can be contrasted to that of imperative languages such as Ada and C/C++, where the primary method of computation is the manipulation of variables. In addition, functional languages also exhibit a number of advanced features, such as mathematically rigorous semantics, polymorphic typing, higher-order functions, unrestricted first-class values and partial application of functions.

For many years, functional languages were of interest only to researchers. ML, for example, was designed as the metalanguage for the Logic for Computable Functions verification system. However, in the past few years researchers have initiated several attempts to demonstrate the efficacy of these languages in the development of real-time systems, systems-level programming and/or rapid prototyping.

As noted in both [HaL94] and [HuJ94], the application of functional languages to large, real-world problems shows great promise due to the flexibility and structure of functional languages. In one prototyping study [CHJ94], the functional programming language Haskell was shown to be significantly superior to both C and Ada in ease of programming for a real-world geometric server application. The primary factors in the success of Haskell appear to be the effective use of polymorphic, higher-order functions, partial application of functions and functions as first-class values.

Implementing the advanced features of these is challenging. To support polymorphic functions, an implementation of a language must adopt some form of data representation that allows a given computational object to assume values of many different types. For example, a compiler may not statically assign the result of a polymorphic function to a floating point register if that result may be of type other than float. By the same token, if the result *is* of type float, it may be inefficient to represent it uniformly and place it in a general purpose register.

Various language implementations have adopted different methods to deal polymorphic functions. For example, SML/NJ uses a uniform data representation (heap-allocated pointers) for all objects. While the scheme works, the need to continually reference and dereference pointers is obviously inefficient. Other languages use different schemes.

The focus of this thesis is to review several promising methods used to efficiently implement polymorphic functions and to propose possible methods for the efficient implementation of Polymorphic C.

B. CONCEPTS AND DEFINITIONS

We begin by providing some background information.

1. Type

Types in programming languages loosely correspond to sets. In set theory every entity is either an element of a set or is a set of either elements or other sets. When

considering the universe (i.e., the set of all sets) within the context of a given domain it is natural to organize it in different ways for different purposes. Types arise naturally in such a classification effort as sets of entities which exhibit common usage and behavior.

In computer science, a *type* is a set of computational objects with uniform behavior. For example, any object of type integer can be expected to observe the total ordering one expects from the set of integers. Likewise, any object of type even_integer (a sub-type of type integer) can be expected to be congruent to 0 (mod 2). Declaring an object to be of a certain type is a declaration of membership in some appropriate set or subset of interest and, indirectly, a declaration of the behavior of that object.

2. Monomorphic / Polymorphic Type Systems

Functions and procedures in conventional languages such as Ada are *monomorphic*, meaning that each can be called with exactly one type. To illustrate, consider the Ada implementation of the integer identity function shown in Figure 1.

```
function identity(value: integer) return integer is
begin
  return value;
end;
```

Figure 1. Monomorphic Identity Function in Ada.

This function operates on an integer and returns an integer and is itself a computational object which can be typed. In this case, it is of type `int -> int`, read as "mapping from an integer to an integer". As a result, the compiler will allow this function to be applied only to objects of type `int`. If an equivalent function was required for a different type (e.g., floats), a separate identity function would have to be written and compiled for that type.

In this case, however, the behavior of the function is entirely independent of the type of it's actual parameter. This is the case in many useful functions. One frequently cited example is a linked list of records, where the act of appending a record to the end of the list is entirely independent of the types of it's various fields, except for the one field that contains a "next" pointer. Another example is a function which reverses the order of the elements in an array; the functionality provided by the function is independent of the type of the array elements.

By contrast to monomorphic languages, other languages such as ML allow functions to have more than one type; these languages are said to be *polymorphically typed*, or *polymorphic*. The identity function, and an application of it, can be written in ML as shown in Figure 2.

```
val identity = (fn x => x);
val three = identity(3);
```

Figure 2. Polymorphic Identity Function in ML.

The syntax in this case is straight forward; the identifier `identity` is bound to the function which takes a parameter `x` and returns `x`. This function is a mapping from an object of any type to an object of the same type. The type of the function is written as α -> α, where α is a *type variable* representing any type. This function can therefore be applied to any type and needs to be written and compiled only once.

Also of note in Figure 2 is the complete absence of explicit type information. Nonetheless, ML's type inferencing system is able to correctly deduce the type of the value assigned to `three`. Since the function is of type α -> α, and the actual parameter is of type `int`, then α must equal `int`. That means, in turn, that the return value must be of type `int` and `three` must be of type int, since that value is being assigned to

`three`. This sort of type inferencing is not necessarily unique to polymorphic languages.

3. Strongly Typed Languages

One of the primary goals for anyone designing or implementing a language, be it monomorphic or polymorphic, is to prevent a class of error known as *type violations*. A simple example is given in the ML program shown in Figure 3.

```
val successor = (fn x => x + 1);
val three = successor(3.0);
```

Figure 3. Example of Type Violation.

In Figure 3, the integer successor function, `successor` is defined. Because the right-hand operand of the addition operator is of type `int`, the type of `successor` is inferred by the ML type inference system as being of type `int -> int`. This function, then, is monomorphic; it operates only on integers.

However, the subsequent call to successor attempts to pass to `successor` a value of type `real`. This is a type violation; the program's behavior would be unpredictable if a compiler were to allow such errors in the general case. It is unclear, for example, what the successor function should do if passed an object of type `list` or, for that matter, of type `automobile`. Languages that strictly avoid such type violations are said to be *strongly typed.*

4. Static vs. Dynamic Typing

To prevent type violations, one can impose a static type structure on a program. Types are associated with all expressions (i.e., constants, operators, variables and

functions) in the program. Subsequent analysis of the program can then determine whether type violations might arise during execution.

Sometimes, however, binding expressions to specific types at compile time is too restrictive. This is certainly the case in a polymorphic language, where a given expression (e.g., a polymorphic function) might assume an arbitrary number of types during program execution. In this case, a polymorphic language might only require that expressions are guaranteed to be *type consistent*. For example, if a particular application of a polymorphic function returns an integer, then that return value should ultimately be assigned only to expressions of type integer.

If the type of each expression can be deduced, or type consistency confirmed, at compile time (i.e., statically) the language is said to be *statically typed*. This a useful property for reasons of efficiency. For example, if the compiler can deduce that the value returned from function `foo` is of type `real`, it can place that value in a floating point register vice a general purpose register. Subsequent floating point operations on that value can be performed without first moving the value to an appropriate register.

On the other hand, some languages, primarily object-oriented languages such as Smalltalk, adopt a policy where only the values are assigned a unique type. Variables and parameters may take values of different types at different times. Because of this, the values of operands must be checked immediately before the execution of any operation. Such languages are said to be *dynamically typed*. [Wa90].

A language may remain strongly typed regardless of whether it is statically or dynamically typed. The decision to make a language statically or dynamically typed is a design decision orthogonal to making it strongly typed (which is always preferable) and is beyond the scope of this thesis. Polymorphic C is a strongly and statically typed language.

5. Classification of Polymorphic Forms

Cardelli and Wegner have classified several varieties of polymorphism [CaW85]. In their scheme, there are two major types of polymorphism, *universal* and *ad hoc*, each of which is further subdivided. This classification scheme is described below.

a. Universal Polymorphism

Functions which exhibit universal polymorphism will generally work on an infinite number of types. Such procedures are *universally quantified* on the types of their arguments. In other words, expressing the type of the polymorphic identity function as being of type $\alpha \rightarrow \alpha$ is merely a short-hand way of stating $\forall\alpha.(\alpha \rightarrow \alpha)$, which is read as "for all values of type α, the function accepts a parameter of type α and returns a value of type α". For this reason, formal parameters of type α are often referred to as *quantified* parameters.

Stated in terms of implementation, a universally-polymorphic procedure will execute the same code regardless of type.

(1) Parametric Polymorphism. In parametric polymorphism, a polymorphic function has an implicit or explicit parameter which determines the type of the argument for each application of that function. Functions that exhibit this form of polymorphism are called *generic* functions. These functions generally do the same sort of work independent of the argument type. The polymorphic identity function is one example. The list reversal function shown in Figure 4, is another, as the work performed by the function is independent of the type of the list elements.

```
fun reverse ([]) = [] |
    reverse (H::T) = reverse(T)@[H];
```

Figure 4. List Reversal Function in ML.

It is worth noting that Ada's generic functions are a special case of parametric polymorphism. The ML list reversal function is compiled only once and will then operate correctly on lists of any type. An equivalent generic Ada function is not directly executable; rather, it must be instantiated statically for each type of interest creating, in effect, a set of functions with identical functionality but each operating on lists of a specific type.

Function templates in C++ are also a special case of parametric polymorphism. They are slightly different from Ada's generic functions, however, in that the instantiations for parameters of different types are performed implicitly (i.e., by the compiler vice the programmer). At run time one version of the function exists for each parameter type of interest, as in Ada.

(2) Inclusion Polymorphism. Inclusion polymorphism was introduced to model sub-typing and inheritance. As such, it is the type of polymorphism generally referred to when discussing object-oriented languages. In this form, an object can be viewed as belonging simultaneously to many different types, or classes.

In particular, an object of a derived class can be used whenever an object of a base (ancestor) class is expected. For example, the integer 17 can be viewed simultaneously as a prime integer, an odd integer and an integer between 10 and 20. While inclusion polymorphism is interesting in it's own right, Polymorphic C does not support object oriented features such as classes and inheritance and, so, the issue is tangential to the thrust of the discussion.

b. Ad Hoc Polymorphism

Ad hoc polymorphism is obtained when a function works on several different types but may behave in different, perhaps unrelated, ways for each type.

(1) Overloading. In overloading, the same identifier is used to denote different functions. Any ambiguity is resolved explicitly or implicitly based on

the context of the function call. As such, it is purely a syntactic convenience for the programmer.

A pervasive example is the use of the "+" operator. It denotes integer addition, floating point addition and, in some languages, string concatenation. In languages which allow the programmer to overload predefined operators, it could potentially mean anything at all. In any case, in successive applications on values of different types, a separate monomorphic function tailored to that type is invoked to do the work.

It should be noted at this point that the generic functions found in Ada, noted earlier to be a special case of parametric polymorphism, can also be considered as simple overloaded functions. The precise classification (if one is required) depends on one's point of view. At the source code level, generics exhibit parametric polymorphism; one piece of source code suffices for an unlimited number of types. At the object code level, generics exhibit polymorphism based on overloading; a separate piece of code is executed for each type, depending on context.

(2) Coercion. Coercion allows the programmer to omit semantically necessary type conversions; the required conversions are inferred by the compiler and inserted into the code. For example, in writing the C code `char b = 'a' + 1;` the programmer would be exploiting the fact that the machine representation for characters (the familiar ASCII mapping) can be meaningfully interpreted as a integer and that the result of the integer addition can, in turn, be meaningfully re-interpreted as a character. The same result could have been achieved by making the coercions explicit, as follows: `char b = (char)((int)'a' + 1);`

As will be seen in later chapters, explicit coercion, also known as *type casting*, is an important tool in the implementation of parametric polymorphism. This point is worth making early. The concept of parametric polymorphism and the implementation of parametric polymorphism are distinct issues.

C. THE ROLE OF POLYMORPHISM IN SOFTWARE ENGINEERING

Apart from pure theoretical interest, polymorphic functions have some pragmatic utility to the field of software engineering, particularly when dealing with large organizations and/or large programming projects. The motivation for polymorphic functions are discussed here in the context of two software engineering goals - program correctness and code reuse - and the common obstacle to each.

1. Goals

The need for polymorphic expression in programming languages derives from two important but conflicting goals in the field of software engineering; the ability to prove statically the correctness of a program and the ability to reuse programs or program segments which are known to be correct.

a. Program Correctness.

Much of the work of proving a program correct has nothing to do with language design or implementation. Certain classes of errors (e.g., faulty requirements specification, errors in logic) cannot be addressed easily, if at all, at the level of language design. Other classes of errors, however, such as type violations, can be detected and prevented.

The compiler for a statically typed language can evaluate a program and guarantee that all expressions are type consistent. In a large programming effort this facility is extremely beneficial as separate programmers inevitably introduce new types and/or new variables,. The cost of discovering a type error at run time can be several times as expensive as discovering it at compile time, and is even more expensive if discovered after product delivery.

b. Code Reuse

The desirability of code reuse - the ability to write routines for a potentially unlimited set of applications - is obvious in the context of a large organization

and/or a large software development effort. Of particular interest are those routines which can be reused when new data types are defined.

An Ada generic sort routine serves as a slightly anemic example of this ability. Ada generics serve merely as templates for the construction of specialized executable code and are, as such, reusable only at the source code level. To use such a routine, the programmer must specifically instantiate a specialized version of it for each data type of interest.

Thus, while there will exist some improvement in programmer productivity, the executable may contain many sections of machine code with identical functionality. In addition, each of these sections would have to be re-compiled for each project. A solution whereby both source and machine code could be reused without duplication - and perhaps without recompilation - would be preferable.

2. Obstacles

Static typing tends to prevent code reuse while reusable programs are harder to statically type check. A monomorphic routine to sort integers, for example, is easy to type check but could not subsequently be used to sort strings. At the same time, in the absence of a strongly typed polymorphic type system, a reusable routine to sort elements of an unspecified type might easily be used (misused) on almost any data structure, however inappropriately.

Polymorphic type systems try to reconcile these two goals by providing all of the type safety of a statically-typed monomorphic language and most of the code re-use flexibility of an untyped language [Car84].

D. C/C++

Despite the many powerful features of functional languages, they are unlikely to fall into widespread general use. Languages such as C are extremely popular and already very capable in the domain of real-world, systems-level programming. It is unlikely that a significant number of organizations would discard the enormous investment of

resources in tools and programmer training in favor of any current functional language. A more pragmatic approach might be to incorporate the results of research in functional languages into imperative languages such as C and C++.

Specifically, one might improve C/C++ in the following ways: (1) more rigorous type checking, (2)more robust polymorphism and (3) the implementation of first-class, higher-order functions.

1. Type Checking

Despite claims to the contrary [Str91], C/C++ too often behaves as a weakly-typed language. C incorporates an unrestricted coercion mechanism, where by a value of one type is automatically interpreted as belonging to another type whenever necessary and possible. Consider the C++ program in Figure 5.

Here, the function `identity` is the monomorphic integer identity function with type `int -> int`. Function `main` then invokes the identity function with two separate non-integer values. The output is as follows:

```
X:  2
Y:  65
```

Clearly, this program is incorrect. The value 2.9 was coerced to an integer representation (by truncation), while the character 'A' was coerced to it's integer ASCII representation. An ardent C/C++ programmer might argue that implicit coercion of this form ("in the right hands", of course) is a feature vice a deficiency. However, some programs are not "in the right hands" and the detection of precisely this class of error is why type checking has been so useful in programming languages.

2. More Robust Polymorphism

C is a monomorphic language. C++ is termed a polymorphic language, but the form of polymorphism is more anemic than found in most functional programming languages. C++ provides function and class templates and, like all other object-oriented languages, allows inheritance between classes. These are legitimate forms of polymorphism, however, extending C++ to allow for ML-style parametric

polymorphism would allow for much greater flexibility in coding and far greater ease in code reuse.

```
int identity(int value) {return value;}

void main()
{
  int X = identity(2.9);
  cout << "X: " << X << endl;

  int Y = identity('A');
  cout << "Y: " << Y;
}
```

Figure 5. Application of Monomorphic Identity Function in C++.

The traditional method of achieving parametric polymorphic expression in C/C++, namely the use of pointers to type void, is instructive. In Figure 6, the identity function, `id`, is declared as one which takes a pointer to void and returns that pointer as a result. As such, `id` can be viewed as a polymorphic function of type $\alpha \rightarrow \alpha$. Before calling the function, it's parameters are referenced and the resulting pointers are cast as pointers to void. Following the call, the returned pointer is cast to it's proper type and dereferenced to obtain the required value.

Two points are worth making in advance. First, the pointer returned from `id` can be interpreted in any way, including an inappropriate way, as seen in the last call to `id` in Figure 6, where the result of passing a character to the identity function is cast as a float.

Second, this general mechanism - converting an object to some uniform representation before a function call then carefully reconverting it to its regular form after the call - is exactly the conceptual scheme (with some refinements) used by many

functional programming languages to implement parametric polymorphism. The implementations of these languages eliminate the need for explicit casting on the part of the programmer.

```
void* id(void* x){return x;}

void main()
{
  int   i = 123;
  char  c = 'A';
  float f = 123.4;

  int   intResult   = *(int*)   (id((void*)&i));
  char  charResult  = *(char*)  (id((void*)&c));
  float floatResult = *(float*)(id((void*)&f));

  float badResult   = *(float*)(id((void*)&c));
}
```

Figure 6. Emulating Parametric Polymorphism in C.

3. Higher-Order Functions and First-Class Functions

Functional languages treat functions as first-class values. As such, they can be passed as parameters, returned as function results, be included in composite values, and so forth, and are referred to as *first-class functions*. A *first-order function* is one whose parameters and result are non-functional. By contrast, a *higher-order function* is one which can take another function as a parameter and/or return a function as a result. [Wa90].

C/C++ treats functions as first-order, second-class values and can be improved by allowing functions to be expressed as higher-order, first-class values. Consider the ML code in Figure 7 which shows a common use of higher-order, first-class functions to apply a given function to each element in a list.

```
fun map (f, nil) = nil |
    map (f, head::tail) = f(h)::map(f, tail);
```

Figure 7. Example of Higher-Order Function in ML. From [St92].

The function `map` is very interesting and is indicative of the style and power of functional languages. First, though, a brief description of the syntactic elements is needed. The keyword `nil` signifies the empty list. The symbol "`::`" is the list catenation symbol with the identifier to the left of the symbol representing the element at the head of the list and the identifier to the right representing the rest of the list. In other words 1::[2,3] = [1, 2, 3]. The symbol "|" simply means "or".

The function `map`, then, is defined as the function that takes as a parameter a function and a list and returns a list. The list may be empty, in which case it is returned. The returned list is computed by applying the supplied function to the element at the head of the list and appending the result to the front of a new list returned by a recursive call to `map` on the tail of the list.

Several things are of note in this example. The first is the relative ease with which a function of this sort can be expressed once one is familiar with the style and syntax. This occurs relatively naturally as a result of the Prolog-style pattern matching used in ML expressions.

The second is the complete absence of explicit type information. In this case, the function is both completely polymorphic and completely type consistent. Based solely on the structure of the function, the ML type system is able to deduce that `map` is of type (α `->` β) `*` α `list ->` β `list`. As such, it takes as arguments a function of type α `->` β and a list of type α `list` and returns another list of type β `list`.

An application of function `map` might be as shown in Figure 8 with the result `newList = [2, 3, 4]` of type `int list`.

```
fun successor (x) = x + 1;
val intList = [1,2,3];
val newList = map(successor, intList);
```

Figure 8. Application of Function Map.

The function `successor` is the integer successor function defined in ML's functional notation; it is read as "bind the identifier `successor` to the function which takes a parameter `x` and returns the value `x + 1`. Because of the integer literal in the function body (`x + 1`), the type system is able to determine that `successor` is of type `int -> int`. In the context of the type assignment for map, then, $\alpha = \beta =$ `int`.

In the second line, the identifier `intList` is declared and defined using a list aggregate as a list with elements of type `int` (`[]` are ML's list construction operators); it is assigned the type `int list`.

The application of `map`, then, is of type `(int -> int) * int list -> int list`, which is completely consistent with both our expectations and the quantified type assigned to the function `map`.

In C/C++, however, the only two things one may do with a function are a) call it or b) take its address [Str91]. To avoid the complexity associated with manipulation of lists (which are not a base type in C++) let's consider a simpler example. The ML and C++ versions of the monomorphic higher order function, `HOF`, are shown in Figure 9.

The ML code is relatively straight forward. As in the case of the function `map`, the programmer is allowed to operate at a relatively high level of abstraction.

The C++ code, on the other hand is much more tedious and error prone in that the programmer is not allowed to abstract away from the underlying mechanisms of the language. In this case, since functions are not first-class values, the programmer must explicitly declare a new type, `INT2INT`, representing a pointer to a function of type `int -> int`. Function `HOF` is then defined as one which takes a pointer to a function of type `INT2INT` and an integer. The function is then applied, via the pointer, to the integer.

Of note is that the C version of `HOF` is not truly higher-order since neither of it's arguments are functions; it only simulates the behavior of a higher-order function. Also, although type information has been supplied explicitly throughout the routine, in the end, type errors of the class described earlier (e.g., `int theInteger = 'A'`) are still not prevented due to C's unrestricted coercion mechanism.

ML:

```
fun Double (x) = x * 2;

fun HOF (f, x) = f (x);

val DoubleInt = HOF(Double, 3);
```

C++:

```
int Double(int x){return x * 2;}

typedef int (*INT2INT)(int);

int HOF(INT2INT f, int x){return f(x);}

int DoubleInt = HOF(Double, 3);
```

Figure 9. Comparison of Higher-Order Functions in ML and C++.

Simulating polymorphic higher-order functions equivalent to the ML `map` function are even more difficult in C. Doing so requires frequent referencing and

dereferencing, casting and recasting, in order to achieve the same results as are achieved without effort in a functional language. Specific examples are given later.

One further benefit of higher-order functions is worth mentioning. In addition to accepting functional arguments, higher-order functions can also return functional results. This leads to the potential for huge benefits in expressiveness and programmer productivity. One can imagine setting out to build a library of trigonometric functions, for example. After carefully designing and testing the function `sin(x)`, one could then simply define `cos(x)` as follows: `cos(x) = sin(x+90)`.

Hudak and Jones have reported significant success using this type of approach in large, real-world problems [HuJ94].

E. INTRODUCTION TO POLYMORPHIC C

The following introduction to Polymorphic C borrows heavily from the initial paper on Polymorphic C, [SmVo95].

Polymorphic C is a polymorphic dialect of C. It is designed to be as close as possible, semantically, with the original K&R C [KR78]. As such, it is stack-based with pointers, variables and arrays. Pointers are first-class values and can be explicitly dereferenced. Variables are second-class values and are implicitly dereferenced. It has the same pointer operations as C, namely the pointer dereferencing operator (*), the address of operator (&) and pointer arithmetic.

Unlike C, it incorporates an advanced polymorphic type system similar to those found in functional programming languages and allows first-class, higher-order functions. And, unlike functional languages, the type system also addresses the polymorphic typing of pointers. The combination of these enhancements results in a language with the flexibility of C and the natural, type-sound polymorphism of ML.

To accomplish this result, the designers imposed one key restriction: "The free identifiers of any lambda abstraction must be declared at top level". Informally, this means that any object used by a function must be either local to the function (i.e., bound

to the function abstraction and having a lifetime that ends upon return from the function) or global (i.e., declared at top level).

The internal static variables of C are an example of the first type of violation. They are declared locally but persist after return from the function. Polymorphic C, then, does not support internal static variables; rather, they must be replaced with uniquely-named global variables to achieve the same functionality.

The second type of violation can occur when function declarations are nested. This violation cannot occur in C but, for the sake of completeness, Figure 10 gives a sample Ada program which highlights this type of violation. Inside `foo` a variable `foo_x` is declared and initialized. Then, also inside `foo`, the function `bar` is declared. Function `bar` uses the variable `foo_x` (which is visible to it) in it's body.

In this case, the variable `foo_x` is free in function `bar` but is not global. This is an example of the second type of violation of the restriction on lambda abstractions in Polymorphic C.

Beyond these simple examples, the restriction on lambda restrictions has only one other consequence. Most functional languages allow partial application (currying) of functions; Polymorphic C does not allow curried functions. Since this issue is tangential to the implementation of polymorphism, it is included here only for completeness and will not be discussed further.

The only other issue of immediate interest deals with the passing of parameters. In most imperative languages, as in C, the formal parameters of a function are local variables. In Polymorphic C they are constants.

```
procedure foobar is
   function foo return integer is
      foo_x: integer := 123;
      function bar return integer is
      begin
         return 2 * foo_x;
      end;
   begin
      return bar;
   end;
begin
   put(foo);
end;
```

Figure 10. Violation of Polymorphic C Restriction on Lambda Abstractions.

II. IMPLEMENTING POLYMORPHISM

Chapter I, Section B, discussed the various types of polymorphism, which is generally categorized as either universal or ad hoc. Universal polymorphism is further categorized as either parametric (generic) polymorphism or inclusion polymorphism; ad hoc polymorphism is further categorized as either overloading or coercion.

This chapter begins with a review of three techniques for implementing polymorphism: textual polymorphism, uniform polymorphism and tagged polymorphism. These terms are unfortunate in that they seem to be additional forms of polymorphism but they are not. Rather, they are general techniques for *implementing* polymorphism.

As seen earlier, the precise meaning of polymorphism in a system is interpreted with respect to a given level of abstraction. For example, Ada's generic functions could be viewed a special form of parametric polymorphism because the source code was independent of the type of the parameters and return values. At the same time, they could be viewed as a special case of overloaded functions since the underlying machine code was dependent on type.

When discussing the implementation of polymorphism, it is necessary to first define the level of abstraction at which one is operating. [MDCB91] takes a pragmatic approach to this issue by defining the possible levels of abstraction based on whether or not the source code, machine code and/or underlying store representations are dependent on data type. Table 1 outlines the sensible combinations.

In order for any polymorphism to exist, the source code of the program must contain expressions which are independent of data type. If this is the limit of polymorphic expression in an implementation, it is termed *textual polymorphism*. If both source and machine code are independent of data type, but store representations may be

	Textual	Tagged	Uniform
Source Code	Independent	Independent	Independent
Machine Code		Independent	Independent
Store Representation			Independent

Table 1. Levels of Abstraction for Implementing Polymorphism.

different for each data type, it is termed *tagged polymorphism*. If source code, machine code and store representations are all independent of type, it is termed *uniform polymorphism*. Each of these is discussed separately in following sections.

A. TEXTUAL POLYMORPHISM

In order for any polymorphism to exist, the source code of the program must contain expressions which are independent of the type of data. If this is the limit of polymorphic expression, it is termed textual polymorphism. A generic function in Ada, a C++ template and an overloaded "+" operator are all examples of this form of implementation.

Since textual polymorphism applies only to source code, the compiler is free to generate optimum code and optimum data representations for each of the specializations of the function. For example, the polymorphic identity function λx . x might be invoked with parameters of three different types within a given compilation unit (e.g., `x:int`, `x:string`, `x:empRec`). A textual polymorphic implementation would generate three specialized functions, `λ(x:int).x`, `λ(x:string).x` and `λ(x:empRec).x`; the function that is actually called would be determined statically from the context of the call.

The space overhead associated with this implementation technique can be quite severe. Within a given compilation unit there are significant space inefficiencies if

several large procedures must each be specialized for many different types. Even worse is the case of separate compilation, where the number of types is not known statically.

[MDCB91] notes that, within a compilation unit there is an upper bound on the number of specialized forms that must be generated. In theory, this number could be quite large. If a function takes p quantified parameters, each of which might be of n possible types, there are n^p possible specializations. For example, the function `λ(x:α, y:β, z:χ).[x, y, z]`, which takes three quantified parameters `x`, `y`, and `z` of types α, β, and χ, respectively, and returns a record of three fields containing the values of those parameters, would have n^3 possible specializations, where n is the number of types in the system.

In practice, the compiler would have to generate, at most, one specialized form of the procedure for each static call. Some of these would share representations, further limiting the number of specializations required. However, this would not be the case for separate compilation. Since the context of the call would not be known statically, the compiler would have little choice but to generate all possible forms of specialization.

The complexity of the problem can also become significantly greater in the case of conditional function calls or in the case where a polymorphic function are passed as parameters to other functions. Consider the Napier88 code of Figure 11.

A polymorphic procedure, `first`, quantified on types `a` and `b`, is defined as one which takes as parameters `W` of type `a` and `X` of type `b`. The ellipses indicates some arbitrary procedure body. Then another polymorphic procedure, `second`, is similarly defined; it is quantified on types `s` and `t`, and takes two parameters, `Y` and `Z` of those types, respectively. Inside the body of `second` exists a conditional. Depending on the truth value of `condition`, the procedure `first` is called in one of two ways.

```
let first = proc[a, b](W:a; X:b) ...
let second = proc[s, t](Y:s, Z:t) ...
     if <condition> then first[s, t](Y, Z)
                    else first[t, s](Z, Y)
```

Figure 11. Exponential Expansion of Code. After [MDCB91].

In this code fragment, for each call of `second`, there are two possible calls of `first`. Since the truth value of `condition` is not known statically, two specializations of `first` are required for each specialized call to `second`. The total number of specialized forms is found by multiplying the number of different specializations in the call chain. Introducing the procedure `third`, shown in Figure 12, would cause four specializations of `first` to be generated for each specialization of `third` (two specializations of `second` and, for each specialization of `second`, two specializations of `first`).

```
let third = proc[x, y, z](X:x, Y:y, Z:z)
     if <condition> then second[x, y](X, Y)
                    else second[y, z](Y, Z)
```

Figure 12. Multiplicative Expansion of Code Due to Call Chain.

Passing polymorphic procedures as parameters also introduces a multiplicative growth in code, though to a lesser extent. The Napier88 code fragment of Figure 13 illustrates this.

```
let id  = proc[u](x: u -> u); x

let foo = proc[s](y:s; bar: proc[t](t -> t) -> s)
              bar[s](y)

let oneTwoThree = foo[int](123, id)
```

Figure 13. Multiplicative Growth of Code Due to Polymorphic Higher-Order Functions.

The syntax and structure of the Napier88 code in Figure 13 requires some explanation. Function `id` is defined as a polymorphic function quantified on type `u`; it accepts a parameter, `x`, of type `u` and returns a value of type `u`, that value being `x`.

Function `foo` is defined as a polymorphic, higher-order function, quantified on type `s`; it accepts as parameters a value, `y`, of type `s`, and a polymorphic function, `bar`, quantified on type `t`, which accepts and returns a value of type `t`. The function `foo` returns a value of type `s`, that value being the result of applying `bar` to `y`.

Lastly, the variable `oneTwoThree` is declared and assigned the value resulting from applying `foo`, specialized to type `int`, to the value `123` and the function `id`. As this example demonstrates, for each specialization of `foo`, there must be a corresponding specialization of `bar`.

Even in cases where the space complexity associated with textual polymorphism can be accepted, there are additional considerations. For example, the code generation scheme discussed above will not work when polymorphic procedures are first-class values [MDCB91].

When functions are first-class, functions may be assigned and substituted for one another if of the same type. In the examples considered thus far, all functions were statically defined; that may not be the case for first-class functions. Consider Figure 14 where a first-class polymorphic function, `foo`, is declared as one of two possible functions. If at run-time `condition` is true, the identifier `foo` is bound to the

polymorphic function, quantified on type `t`, which returns the first of it's two actual parameters; otherwise it is bound to the polymorphic procedure which returns the second of it's two parameters. The type specializations are known statically and the compiler can determine that function `foo` has to be specialized prior to call, but it cannot determine statically which of the two possible bindings to specialize.

```
let foo = if <condition> then proc[t](a, b: t -> t); a
                         else proc[t](a, b: t -> t); b
let fooint = foo[int](1, 2)
let fooreal = foo[real](1.0, 2.0)
```

Figure 14. Example of First-Class Polymorphic Procedure.

One solution might be to generate specializations for both potential bindings and have the run-time system pass pointers the proper code. This solution, though, suffers from the same space complexity considerations discussed previously in the context of static specialization.

Another solution might be to invoke the compiler dynamically whenever a new specialization is required. In other words, the function `foo` would not be compiled at all until the proper binding has been determined. This solution could make the call to `foo` very slow and inefficient.

In summary, textual polymorphic implementations demonstrate the ability to produce optimum code and data representations for each application of a polymorphic procedure and can be used to implement both ad hoc and uniform polymorphism. Major disadvantages, however, include a potentially large amount of generated code for each application of a polymorphic procedure and an inability to deal efficiently with first-class polymorphic procedures.

B. UNIFORM POLYMORPHISM

If both the source code and the machine code for an implementation are independent of the data types being manipulated, and if the store representation is uniform for all data types, the polymorphic implementation is termed uniform polymorphism. ML uses this implementation technique. The function `reverse` of Figure 4 is written only once, compiled only once, and operates on any list, independent of the data type of it's elements.

The major trade-off for this form of implementation is that for uniform code to function correctly on all data types, the values for all data types must have a uniform representation (i.e., must all be of the same size). Using the list reversal function as an example, the underlying machine code must eventually swap around bytes in storage to perform it's work; it must, as a minimum, know how many bytes to swap. If the code is to work uniformly on all data types (e.g., lists of integers, lists of reals, lists of strings, even lists of lists), all data types must be represented by the same number of bytes.

This uniform representation may not be optimal for some types. For example, if a store size of one byte is used, it becomes difficult to implement double word floating point numbers. If a store size of eight bytes is selected, then the implementation of short integers and characters becomes inefficient.

A second complication arises in the implementation of compound data types. Because a fixed-sized data representation is required for uniform polymorphism and because compound data types can be arbitrarily large, pointers are the only efficient way to refer to them. The alternatives are clearly more inefficient or impossible: choosing a uniform representation large enough to hold any arbitrarily large data structure is impossible; choosing a representation large enough to hold the largest object in a system would be possible but would be extremely inefficient and, in the case of separate compilation, might be unknown. A third alternative, selecting a representation of some suitable arbitrary size and requiring all data objects to "fit" might also be possible but would add undesirable complications at the time of creation or reference.

If objects of compound data types are to be represented by pointers, and the representation is to be uniform for all data types, then all data objects must be represented by pointers. This adds a level of indirection to the implementation of every object in the system, including simple scalar objects such as integers. In an environment with implicit garbage collection, it also necessitates garbage collection on unused objects.

Moreover, the overhead arising from uniform data representation will exist in the system even if polymorphic expression is not required in a particular module. The mere potential for such expression is sufficient to invoke the requirement.

In summary, uniform polymorphism is relatively easy to implement and is relatively efficient with respect to space. However, all objects must be represented in a uniform, non-optimal form irrespective of the degree of polymorphism in the system.

C. TAGGED POLYMORPHISM

In some systems the source code and the machine code are both independent of the data type being manipulated but the data representations, and possibly the behavior of the program, are nonuniform for different data types. Such systems are instances of tagged polymorphism. In a tagged polymorphic implementation, each data item is tagged with some form of type information. The machine code is constructed to use this type information to determine dynamically which of several type-dependent instructions to execute.

Examples of this form of polymorphism can be found in the implementation of inclusion polymorphism in many object-oriented languages. In the language Actor, for example, each object contains a method dictionary, with method names as keys and pointers to methods as values, which served as an address map for it's methods. The static machine code for searching the method dictionary is the same for all methods in all objects but the dynamic behavior of the system depends on the value returned from the search. In this case, the method dictionary is effectively a tag.

Tagged polymorphism, used in this fashion, can be seen as a means of implementing a built-in form of ad hoc polymorphism (overloading and coercion). The

common methods of a set of objects derived from a common super-class are, in a sense, simply overloaded functions. In the case where a method of the super-class is not redefined by a sub-class, the method of the super-class is invoked; in effect, the object of the sub-class is coerced to an object of the more general super-class.

Another example, given in [MDCB91], is the tagged architecture of the Burrough's B6500. That system included several polymorphic machine instructions, such as plus, minus, times, etc. Data was tagged according to it's type and when an a plus operation, for example, was issued, the processor would inspect the tag and perform either integer or floating point addition, depending on the value of the tag.

Tagged polymorphism can also be used to implement parametric polymorphism. However, it may be unacceptable to map an infinite number of types onto a finite number of tags. Still, to the extent that such a mapping is feasible for a given system, parametric polymorphic expression is possible.

In summary, tagged polymorphism can implement ad hoc, inclusion and parametric polymorphism. It is efficient with respect to the amount of generated code and can operate with non-uniform data representations. However, the polymorphic expressions are built-in and, because all data objects must be tagged and those tags frequently inspected, the tagging system must be very efficient.

III. IMPLEMENTATION OF POLYMORPHISM IN NAPIER88

This chapter presents a case study of the polymorphism implementation techniques used in Napier88. Chapter IV presents a case study of the polymorphism implementation techniques the recently proposed extensions to ML [Le92]. Both are functional languages with full polymorphic higher-order, first-class functions. While neither of the implementations studied here are directly transferable to Polymorphic C, many of the concepts and motivations behind these techniques are useful in developing an implementation strategy for Polymorphic C.

Napier88 uses a variant of the tagged polymorphic implementation technique using procedure closures to capture type information. The primary thrust of the approach is based on the requirement that only polymorphic procedures should pay the penalty for polymorphic expression and the observation that only the polymorphic expressions within polymorphic procedures need exhibit uniformity of behavior. Outside a polymorphic procedure, this uniformity is not necessary.

All data objects are stored in their system-dependent, optimal representations (called *concrete* form) and can be manipulated by monomorphic procedures in that concrete form. Objects which are passed to quantified formal parameters of a polymorphic procedure are coerced to a uniform representation (e.g., pointers) on entering the polymorphic procedure and coerced back to their concrete representation on exit from the procedure. Within the polymorphic procedure, the objects are manipulated using their uniform representation.

The following discussion closely follows [MDCB91] which contains a complete description of the implementation of polymorphism in Napier88.

A. POINTS OF CONVERSION

As stated previously, objects with concrete representation must be coerced to uniform representation if passed to the quantified formal parameter of a polymorphic procedure. This coercion might be performed either before or after the call.

For a programming language that allows the combination of first-class procedures and type specialization without call, the compiler is unable to determine statically whether a procedure being called is polymorphic or monomorphic. This is the case in Napier88. Consider the Napier88 code in Figure 15.

```
let first = proc[t](a, b: t -> t); a

let either = if <condition>
             then first[int]
             else proc(a, b: int -> int); b

let two = either(2,3)
```

Figure 15. First-Class Procedure and Specialization Without Call. From [MDCB91].

In this code, the procedure `first` is a polymorphic procedure which returns the first of two quantified parameters. At the time of the call `either(2,3)`, if `condition` is true the identifier `either` is bound to the first-class polymorphic procedure `first`, specialized for type integer, which returns the first of two integer parameters. Otherwise, it is bound to a monomorphic procedure of type `int -> int` which returns the second of two integer parameters.

In either case, the compiler does not know statically whether the procedure being called is polymorphic or monomorphic and therefore cannot determine statically the proper representation (uniform or concrete) for the actual parameters. If a conversion is performed and the function turns out to be monomorphic, therefore expecting concrete actual parameters, the results are unpredictable. This problem could be solved by

compiling monomorphic functions to accept uniform values and convert them as required, but that solution violates the requirement that only polymorphic functions suffer polymorphic overhead.

As a result, the conversion to uniform representation must be delayed until after the call. If the procedure turns out to be polymorphic, the conversion would have to occur within the polymorphic procedure itself.

In general, there are four cases of interest when passing parameters to a polymorphic procedure. These are shown in Table 2 and discussed in the following sections.

1	Concrete actual parameter passed to concrete formal parameter.
2	Concrete actual parameter passed to quantified formal parameter.
3	Quantified actual parameter passed to concrete formal parameter.
4	Quantified actual parameter passed to quantified formal parameter.

Table 2. Passing Parameters to Polymorphic Functions.

1. Concrete Actual Parameter Passed to Concrete Formal Parameter

This case is trivial since there is no polymorphism involved. The compiler is free to generate monomorphic code.

2. Concrete Actual Parameter Passed to Quantified Formal Parameter

In this case, for every formal parameter of a quantified type, the concrete actual parameter must be converted inside the polymorphic procedure to the system's uniform representation and the result - if of a quantified type - must be converted back to it's non-uniform representation on exit. Figure 16 shows an example.

On the calls `first[int](1, 2)` and `second[int](1, 2)`, the first formal parameter `x` is of quantified type, so the first actual parameter, `1`, is converted to the system's uniform representation. The second formal parameter `y`, however, is of

```
let first = proc[t](x: t; y: int); x
let second = proc[t](x: t; y: int); y

let one = first[int](1, 2)
let two = second[int](1, 2)
```

Figure 16. Concrete Actual Parameter Passed to Quantified Formal Parameter.

concrete type, so the second actual parameter, 2, is left in it's optimum, concrete representation. In this regard, the two calls are identical; the first actual parameter is manipulated in the system's uniform representation while the second actual parameter is manipulated using it's concrete representation.

On exit, however, the two procedures behave differently. In the case of the call `first[int](1, 2)`, the return value, 1, is of quantified type and must be converted back to it's original representation. In the call `second[int](1, 2)`, the return value, 2, is of concrete type and, so, need not be converted.

In all instances of this case, specialized code is required to convert from any representation to/from the uniform representation. All other code in the polymorphic procedure is uniform.

3. Quantified Actual Parameter Passed to Concrete Formal Parameter

As seen above, a polymorphic procedure converts objects passed via quantified formal parameters to a uniform representation. If the polymorphic procedure subsequently passes that object to another procedure(be it monomorphic or polymorphic) via a concrete formal parameter, the object must be converted to it's concrete representation prior to the call. Figure 17 shows an example.

Here, procedure `foo` is quantified on type `t` and takes as quantified parameters `x`, of type `t`, and `y`, a procedure of type `t -> t`. Since both parameters are quantified, both are converted to uniform representation on the call to `foo`. The procedure returns a quantified value which is obtained by applying the second parameter to the first.

```
let int_id = proc(x: int -> int); x

let foo = proc[t](x: t; y: proc(t -> t) -> t); y(x)

let three = foo[int](3, int_id)
```

Figure 17. Quantified Actual Parameter Passed to Concrete Formal Parameter.

In this case, however, the second parameter is the monomorphic integer identity function which expects to be passed an integer in concrete form. Therefore, the actual parameter must be converted to concrete form prior to the call and the return value must be reconverted to uniform form after the return. Prior to returning from `foo`, the result is again converted to concrete form.

4. Quantified Actual Parameter Passed to Quantified Formal Parameter

Polymorphic procedures expect to convert their parameters to uniform representation on entry. This means that a polymorphic procedure which passes an object with uniform representation to a procedure via a quantified parameter must pass that object in it's concrete representation. The behavior is the same as given in the third case; the object must be converted to concrete form prior to the call and reconverted following return. Figure 18 illustrates this case.

Here, the polymorphic procedure `id2` is the polymorphic identity function, quantified on type `t`. It's return value is obtained by invoking procedure `id1`, another version of the polymorphic identity function. At the time of the call, id2 is initialized at type `int` and, so, `id1` is also initialized at type `int`. As a result, `id1` expects a parameter of type `int` and, being a polymorphic procedure, it expects convert that integer to uniform representation. It is necessary, then, for `id2` to convert the actual parameter `y` to the concrete representation for integers prior to the call to `id1`.

```
let id1 = proc[s](x: s -> s); x
let id2 = proc[t](y: t -> t); id1[t](y)

let two = id2[int](2)
```

Figure 18. Quantified Actual Parameter Passed to Quantified Formal Parameter.

B. DATA STRUCTURES

The preceding discussion on points of conversion dealt solely with atomic types. The introduction of data structures raises one minor additional issue, best illustrated by an example, shown in Figure 19.

Here, the data structure `tuple` is defined and is given a constructor called `make_tuple`. Both are quantified with respect to their formal parameters. The call `make_tuple[int, int](1,2)` leads to the creation of the ordered pair `(1, 2)`.

This single data structure might be referenced after creation by both monomorphic and polymorphic procedures. If referenced by a polymorphic procedure, the individual fields of the tuple must be referenced using a uniform representation. If referenced by a monomorphic procedure, it's fields must be referenced using the system's concrete representation.

The solution to this problem is to view access to compound data structures as a special case of parameter passing. They are always created and stored using concrete representation to allow access by monomorphic procedures. When they are accessed by a polymorphic procedure a conversion takes place within the polymorphic procedure.

C. NAPIER88 BLOCK RETENTION ARCHITECTURE

If a language does not incorporate block retention, the memory reserved for variable declared within a block may be reclaimed on exit from the block. Languages that support arbitrary higher-order functions, as does Napier88, must incorporate a block retention architecture, meaning that variables declared within a block may persist after

```
type tuple[s, t] is structure (first: s; second: t)

let make_tuple = proc[u, v](a:u; b:v -> tuple[u, v])
     tuple[u, v](a, b)

let this_tuple = make_tuple[int, int](1, 2)
```

Figure 19. Creation of Data Structure by Polymorphic Procedure.

exit from the block. An example might be the internal static variable found in C. The example of Figure 20 illustrates this.

The block defined by `random` contains the declaration of the variable `seed`. If `random` is to work correctly, seed must persist between calls to `random`. If it does not, `random` will return the same value each time it is called. As will be seen later, internal static variables such a seed are important to the Napier88 implementation of polymorphism.

It is also worth noting that Polymorphic C does not support internal static variables. Such variables would have to be declared as uniquely-named global variables. Being global, their lifetime is that of the program, so the requirement for block retention does not apply to Polymorphic C.

```
let random =
begin
     let seed := 2111
     proc(-> int)
     begin
          seed := (519 * seed) div 8192
          seed
     end
end
```

Figure 20. Block Retention. From [MDCB91].

D. IMPLEMENTATION OF ATOMIC TYPES

Earlier discussions with respect to points of conversion made clear that under the constraint imposed by the Napier88 implementors (only polymorphic procedures pay a penalty for polymorphism), polymorphic functions were required to convert their formal parameters to and from concrete and uniform representations. But how is this accomplished?

Clearly, if a polymorphic procedure is to convert a data object from uniform to concrete form prior to return, it must know the concrete form to which it should be converted. This functionality cannot be hard-coded in the polymorphic procedure (e.g., always convert to concrete representation for integers) because the function is polymorphic and must work with values of any type.

The tagged implementation discussed in Chapter II might be used for the conversion but data tagging is contrary to the requirement that only polymorphic functions suffer overhead due to polymorphism; if all data is tagged to support polymorphism, then the entire system, including the monomorphic part, suffers the overhead associated with polymorphism.

The only other solution is for the polymorphic procedure to receive, in some manner, information about the concrete types of it's quantified parameters. One possible solution might be to simply pass type information to the function as a separate parameter. This will work, but the implementors of Napier88 chose an alternative solution which takes advantage of block retention in the language (i.e., internal static variables). This helps avoid having to pass an extra parameter at each function application.

In Napier88, a polymorphic procedure is compiled into one in which the type parameter is represented by an integer in an outer level (envelope) procedure of the same name. The polymorphic executable code is bound to the envelope procedure, with the result that the type tag is contained in its closure. For example, consider the arbitrary

polymorphic function `foo` as defined and applied in Figure 21(a), which would be compiled into that shown in Figure 21(b).

```
(a) let foo = proc[t](x: t -> t); ...

    let a = foo[int](3);

(b) let foo = proc(tTag:int -> proc(α -> α))
                proc(x:α -> α); ...
```

Figure 21. Napier88 Polymorphic Identity Function. After [MDCB91].

In Figure 21(b), `tTag` is an integer encoding of the quantifier's specialization type; it varies for each call. For example, a particular system might have a mapping represented by the following case statement for the concrete type τ:

```
let tTag = case τ of:
             τ = int    :1;
             τ = string :2;
             τ = real   :3;
           default      :0;
```

The symbol α represents, at any type specialization, the concrete type of the quantified type [MDCB91]; in other words, it is a type variable.

For clarification, consider the call `foo[int](3)` of Figure 21(a), which is compiled into two calls. The first is equivalent to `let int_foo = foo[int]`. This creates an envelope procedure of type `proc(tTag:int -> proc(α -> α))` with the type tag for type `int`; the result is a monomorphic procedure called `int_foo`. In this case, `int_foo` is monomorphic code which, for this specialization, happens to have the type tag for `int` contained in an internal static variable. The original polymorphic code for `foo` is dynamically bound to this envelope procedure; thus it has the type tag for `int` in its closure.

The second call is equivalent to `int_foo(3)`, which calls the polymorphic executable code with the concrete parameter `3`. After the call to the polymorphic procedure, the concrete parameter must be converted to the uniform data representation. To do this, a special built-in generic instruction, called *convertToPoly(`tTag`)*, inserted into the body of foo at compile time, is invoked. This instruction uses the type tag contained in the closure of the polymorphic code to convert the concrete parameter to the system's uniform representation. Prior to exit from the polymorphic code, the built-in generic instruction *convertFromPoly(`tTag`)* is executed, again using the type tag in the code's closure to perform the proper conversion. In this way, the polymorphic code can remain uniform for each specialization yet perform conversions to and from concrete representations of any type.

Thus, the call `foo[int](3)` would result in the following chain of events. An envelope procedure, call it `int_foo`, would be created and the integer encoding for type `int` would be placed in a variable, `tTag`, local to `int_foo`. The polymorphic code `id` would then be bound to `int_foo`, causing `tTag` to be visible to `foo`, or, more specifically, to elements within the body of `foo`. Procedure `foo` is then called with the integer value `3`.

Within the body of `id`, the integer must be converted to uniform representation. The first statement in `foo` is an invocation of the generic system instruction `convertToPoly(tTag)(3)`, the result of which is the integer `1` represented in uniform form; assume it is bound to identifier `x`. The polymorphic procedure is then free to manipulate `x` in a uniform manner. Prior to exiting the body of `foo`, the return value must be converted back to concrete representation by invoking the generic instruction `convertFromPoly(tTag)(x)`. The result of that call is returned.

If `foo` is subsequently called with a value of a different type, a new envelope procedure with a different type tag is created and the polymorphic code for `foo` is dynamically re-bound to this new procedure. Thus, from the standpoint of the polymorphic procedure `foo`, the only difference between the call `id[real](3.0)`

and the call `id[int](3)` is the value of `tTag` in its current closure. The code for `id` remains the same; in fact, the exact same machine code is executed in each case. The only difference lies in which type-specific versions of `convertToPoly(tTag)` and `convertFromPoly(tTag)` are invoked at entry and exit.

E. IMPLEMENTATION OF DATA STRUCTURES

As mentioned previously, all data structures are stored in non-polymorphic form; when the fields of a data structure are accessed by a polymorphic procedure, they are converted for use within the procedure and are reconverted when returned to the data structure. This scheme is necessary to allow monomorphic procedures to access the structures normally.

There are two cases where a polymorphic procedure may manipulate a value of a quantifier type that is part of a data structure: (1) when the data structure is passed as a parameter and (2) when the data structure is created within the procedure and returned as its value.

1. Data Structure Passed as Parameter

Figure 22 shows an example of passing a structure with quantified fields to a polymorphic procedure. The procedure `findSize` is defined as a procedure, quantified on types `s` and `t`, which takes as a parameter a structure, `A`, with two fields: `age`, of type `s` and `size`, of type `t`. The procedure returns a value of type `t`, that value being the value in the `size` field of the structure.

```
let findSize = proc[s, t](A:structure(age:s; size:t) -> t)
                  A(size)
```

Figure 22. Passing Structures with Quantified Fields. From [MDCB91].

Clearly, since the size of the concrete representations of the types `s` and `t` are unknown and potentially variable, it would be impossible to compile `findSize` using a constant offset for the field `size`. That information would have to be calculated at run-time and passed to the polymorphic procedure.

As in the case of passing type information to a polymorphic procedure, there are two solutions. The first solution, passing offset information as additional parameters, was excluded from consideration in favor of simply extending the tagging method described previously for atomic types. In addition to a type tag for each quantified formal parameter, a field offset value for each field in each quantified formal parameter taking a compound data type is passed to an envelope procedure. These values are then available to the embedded polymorphic procedure in its closure. Figure 23 shows the compilation of the `findSize` procedure.

```
let findSize = proc(sTag, tTag, ageOffset, sizeOffset:int
                -> proc(A:structure(age:α, size:β) -> β))
                     proc(A:structure(age:α, size:β) -> β)
                       A(sizeOffset)
```

Figure 23. Compilation of Figure 22. From [MDCB91].

Some clarification of the syntax is required. The procedure `findSize` is compiled as a monomorphic procedure which takes as parameters four integers; the first two are the usual type tags representing the concrete types of the fields, the second two are the offsets of those fields. The polymorphic code representing the original findSize procedure is bound to this envelope procedure. It takes as parameters a value for `age` and a value for `size`, both of quantified type, with the type tags and offsets for those fields contained in its closure. Upon call, that polymorphic procedure uses the

information in its closure to convert it's actual parameters to and from uniform form and to index into the structure.

2. Data Structure Created Within Polymorphic Procedure

A second addressing problem occurs when a data structure is created within a polymorphic procedure. The structure must be created in concrete form for non-polymorphic use, however neither the offsets for the fields nor the overall size of the structure are known at compile-time; they depend on the particular specialization of the call.

Once again, offset information might either be passed to the polymorphic function in the form of additional parameters or it may be left in the closure of the embedded polymorphic function in the form of local declarations within it's envelope procedure. If, however, the structure is totally encapsulated by the polymorphic procedure, no offset information would be available at the time of call.

The Napier88 solution is to generate code within the envelope procedure to calculate this information and to leave it in the closure of the polymorphic procedure in the form of local declarations. This code uses another built-in system procedure, the monomorphic procedure `typeSize`, to do this work. Figure 24 shows a polymorphic procedure, `mkPair`, and its compilation using this scheme.

Again, the Napier88 syntax is in need of clarification. The procedure `mkPair` is defined as a polymorphic procedure, quantified on types `s` and `t`, which takes as parameters `first`, of type `s`, and second, of type `t`. It returns a structure with fields `fst` and `snd` of types `s` and `t` respectively. The structure is the result of assigning `first` to `fst` and `second` to `snd`.

As compiled, `mkPair` is defined as a monomorphic function with two integer parameters as type tags. The polymorphic code representing the original `mkPair` is bound to this envelope procedure as usual.

```
let mkPair = proc[s, t](first:s; second:t
   -> structure(fst: s, snd: t))
         struct(fst=first, snd=second)


let mkPair = proc(sTag, tTag:int
   -> proc(α, β -> structure(fst: α; snd: β)))
begin
  let fstOffset = 0
  let sndOffset = fstOffset + typeSize(sTag)
  let structSize = sndOffset + typeSize(tTag)

  proc(first: α; second: β -> structure(fst: α; snd: β))
    struct(fstOffset=first, sndOffset=second)
end
```

Figure 24. Compilation of Polymorphic Procedure mkPair. [From MDCB91].

In the body of the envelope procedure (following the `begin` statement), the offsets and overall size for the structure are calculated using the built-in `typeSize` procedure and the type. Finally, the polymorphic procedure is called and uses the offsets contained in its closure to assign the concrete values, `first` and `second`, to the appropriate addresses within the structure.

F. EFFICIENCY AND OPTIMIZATION

The main advantage to the Napier88 technique is that only values of quantifier type are tagged. There is no overhead for monomorphic procedures and there is no overhead for monomorphic portions of polymorphic procedures.

There are two sources of run time overhead. The first is in the fact that two procedure calls are made for every call to a polymorphic procedure: one to the envelope procedure and one to the polymorphic code. The second is in the calls to the built-in procedures which convert between forms and calculate type size information.

[MDCB91] mentions several possible optimizations which are discussed here briefly.

1. Specialization Through Partial Application

If polymorphic procedures are specialized once then called many times, the cost of creating envelope procedures can be amortized over many calls. For example, if the polymorphic identity function is to be called repeatedly with values of type integer, it would be wise to specialize it once (e.g., `let int_id = id[int]`) and then call `int_id`. This is equivalent to the textual polymorphic approach, except specialization is performed at run time vice compile time.

2. Generate Inline Code

It might be possible, from static inspection of the code within a compilation unit, to generate inline code instead of making polymorphic procedure calls. There is no need, for example, to make the call `let x = id[int](123)`.

3. Use Textual Polymorphism

In the case where very few specializations are required, it may be more efficient to generate pure monomorphic code for each type of interest than to suffer the overhead associated with polymorphism. This approach will not be efficient if the number of specializations and/or the number of quantified parameters is large. It will not work at all in the case of first-class polymorphic functions.

4. Static Analysis

It may be possible to elide unnecessary conversions. For example, if a polymorphic function is declared within another polymorphic function and can never escape the scope of that function, it can be compiled to accept parameters passed in uniform form. This would elide four unnecessary conversions to/from concrete form.

IV. IMPLEMENTATION OF POLYMORPHISM IN ML

A second strategy for the implementation of polymorphism is that used in implementing ML [Le92]. Historically, most implementations of ML have used a uniform data representation, specifically single-word pointers, for all data objects in order to support polymorphic functions. A primary result of [Le92] was to allow for a mixed data representation in ML, thereby improving efficiency. This work has been extended in [ShA95] and [Th95].

The implementation strategy in this chapter is similar in many respects to that of Napier88. On a surface level, it is just a variation on the same theme. Values are stored in concrete form, making monomorphic functions much more efficient in the presence of optimal data representations. Values passed to polymorphic functions via quantified formal parameters must share a common representation; concrete actual parameters must therefore be converted to uniform form.

However, ML differs from Napier88 in several important ways, one of which is critical to the implementation of polymorphism. While Napier88 supports type specialization at run-time, ML does not. This seemingly minor difference allows an ML program to be fully type checked statically which allows much greater freedom in the choice of implementation strategies. Polymorphic C also has these properties.

This chapter is structured very much like the previous chapter. After introducing a few terms, we review the possible points at which conversions to and from uniform representation might be required. We then discuss implementation details for atomic data and compound data structures. The chapter concludes with a discussion of efficiency and optimization issues.

A. BOXED AND UNBOXED VALUES

The proposed implementation technique uses some form of uniform representation of data for use by polymorphic functions. In the discussion of Napier88, we used the terms concrete and uniform form. While those terms are generic and are still applicable, the literature regarding polymorphism in ML introduces some other related terms.

The bit-pattern representing a value on which machine instructions operate is called an *unboxed value*; 32-bit integers, 64-bit long integers, single- and double-precision floating point numbers, etc., are all examples of unboxed values. A pointer to a heap-allocated *box* containing an unboxed value is called a *boxed value*. [PJ91].

Conversions to and from concrete form are performed by a pair of generic operators called *wrap(τ)* and *unwrap(τ)*, where τ is some concrete type. `Wrap(τ)` performs the conversion from the concrete representation of type τ to the uniform representation and is usually implemented by boxing the object. The result of this operation is a data object which is said to be in the wrapped state or, simply, *wrapped*. `Unwrap(τ)` performs the conversion from uniform representation to the concrete representation of type τ, by performing the converse of the wrapping operation on that type; the result is an *unwrapped* object.

At times when context is not important, the terms uniform, boxed and wrapped, and the terms concrete, unboxed, and unwrapped, are roughly synonymous. There are subtle differences, though. In the implementation of ML, wrapping is most often performed by boxing to obtain a uniform representation; unwrapping is generally performed by unboxing to obtain a concrete representation.

For clarity in the examples given in following sections, when there is only one concrete type involved we will drop the type quantifier and use the simpler terms wrap and unwrap vice wrap(τ) and unwrap(τ).

B. POINTS OF CONVERSION

In the discussion regarding Napier88 we showed that for a language which allows both first-class functions and type specialization without call it is impossible to know statically whether or not a procedure being called is polymorphic. Because of this, values being passed via quantified formal parameters could not be converted to uniform representation before the call.

However, ML does not allow partial application of types. All variables are statically bound and their type is known at compile-time. By way of explanation, consider the ML code of Figure 25 (an expansion of the Napier88 code shown earlier in Figure 15.

```
fun first(x,y) = x;
fun second(x,y:int) = y;
let condition = true;
val either = if condition then first else second;
val two = either(2, 3);
let condition = false;
val three = either(2, 3);
```

Figure 25. Effect of Static Binding in ML.

The polymorphic function `first` is defined as one which takes two quantified parameters, `x` and `y`, and returns `x`; it is assigned the type $\alpha * \beta \rightarrow \alpha$. Then the monomorphic function second is defined as one which takes two integers, `x` and `y`, and returns `y`; it is assigned type `int * int -> int`.

Identifier `either` is bound to either the function `first` or the function `second`, depending on the truth value of `condition`. In this example, `condition` has been assigned the truth value `true`, so `either` is statically bound to `first`. As a result, on the call `val two = either(2, 3)` the value 2 is returned.

This binding for `either` is static, depending only on the environment at time of definition. If `condition` is given a new binding as shown in Figure 29 and `either` is called a second time, the return value will not change; the function `either` is still bound to `first`.

Because of static binding, the type inferencing engine is always able to determine the type of a function at compile time. The function may be monomorphic or polymorphic but, unlike in dynamically-bound languages such as Napier88, there is no ambiguity and, especially, there is no ambiguity at run time. As a result, the requirement to convert values from concrete to uniform form before, vice after, the call to a polymorphic, higher-order function does not apply. We are free to adopt either conversion convention.

In fact, [Le92] has adopted the convention that conversions should occur before the call to a polymorphic function. In light of this difference, it is useful to once again consider the high-level issues surrounding the various points of conversion. Recall that there are four cases of interest when passing parameters to a polymorphic procedure. These are shown in Table 3 and discussed in the following sections.

1	Concrete actual parameter passed to concrete formal parameter.
2	Concrete actual parameter passed to quantified formal parameter.
3	Quantified actual parameter passed to concrete formal parameter.
4	Quantified actual parameter passed to quantified formal parameter.

Table 3. Passing Parameters to Polymorphic Functions.

1. Concrete Actual Parameter Passed to Concrete Formal Parameter

As before, this case is trivial since there is no polymorphism involved. The compiler is free to generate monomorphic code.

2. Concrete Actual Parameter Passed to Quantified Formal Parameter

In this case, every concrete actual parameter being passed to a function via a quantified formal parameter must be converted to uniform representation prior to the call and the result, if of quantified type, must be converted back to concrete form after the return.

3. Quantified Actual Parameter Passed to Concrete Formal Parameter

If a polymorphic procedure receives a value in uniform form and subsequently passes that value to another procedure via a concrete formal parameter, the value must be reconverted to its concrete form before the call. In the absence of some sort of trick (e.g., tagging), it is not possible for the conversion to occur before the call; doing so would subvert the polymorphic nature of the calling function. Figure 26 shows an example with unnecessary details omitted.

```
fun polySort(aList, aComparisonFunction) = ...

fun intCompare(x,y: int) = ....
val intList = [3,2,4,1];

val aSortedIntList = polySort(intList, intCompare);
```

Figure 26. Passing Quantified Actual Parameter to Concrete Formal Parameter.

First, a polymorphic sort routine, `polySort`, is declared; it takes as parameters a list of quantified type and a comparison function operating on elements of the list. Then, a monomorphic integer comparison function, `intCompare`, and an integer list, `intList`, are declared and passed as parameters to `polySort`. Because `polySort` is polymorphic and it's parameters quantified, `intList` is converted to uniform representation prior to the call.

The function `polySort` will, in the course of its work, determine whether to swap two list elements by applying the comparison function to two of those elements. The comparison function is monomorphic and, so, expects to receive it's parameters in concrete form. But `polySort` received them in uniform form. An explicit conversion in the body of `polySort`, in this case to type integer, would effectively cause `polySort` to become monomorphic.

There are two common solutions. The first is to incorporate a tagging mechanism, whether that tag is actually embedded in the data type or passed as an additional parameter to the polymorphic procedure. In this case, `polySort` could inspect the tag, invoke the appropriate conversion utility and pass the parameter.

A second solution, and the one chosen in [Le92], is to build a monomorphic envelope procedure that performs the correct conversions and then invokes the called monomorphic function. The polymorphic calling function is free, then, to call the envelope procedure, passing values in uniform form, and to rely on the envelope procedure to perform the proper conversions. Numerous examples are given in following sections.

4. Quantified Actual Parameter Passed to Quantified Formal Parameter

If a polymorphic procedure receives a value in uniform form and subsequently passes that value to a polymorphic procedure via a quantified formal parameter, no conversion is required. The called procedure expects a value in uniform form and will receive it as such. Likewise, no conversion is required on return from the called procedure.

C. IMPLEMENTATION OF ATOMIC TYPES

In the approach introduced in [Le92], the implementation of polymorphism for atomic types is straightforward. Between calls, atomic objects are stored in concrete form. Monomorphic functions are compiled using optimal, system-dependent data representations. Polymorphic functions are compiled using uniform data representations.

Conversions between forms are required in only three instances: before passing a concrete value to a polymorphic function via a quantified formal parameter, after return from a polymorphic procedure if the return value is of quantified type, and when passing a quantified value to a function via a concrete formal parameter. These cases are addressed in turn in the following sub-sections.

The program transformations used to realize these conversions are extremely elegant and powerful. Some final results are given below. A more comprehensive treatment of the transformation, along with the derivations of the examples in this chapter, is provided in the Appendix.

1. Applying Polymorphic Functions to Concrete Values.

As stated in the previous section, when passing a concrete value to a polymorphic procedure via a quantified formal parameter, conversions to and from concrete form occur prior to the call and the results, if of quantified type, are reconverted following return. Figure 27 demonstrates this technique.

```
      fun id(x) = x;
      val one = id(1);
(a)   val one = (unwrap(id(wrap(1))));
```

Figure 27. Applying a Polymorphic Function to a Concrete Value.

In Figure 27, a polymorphic identity function, `id`, is defined. The subsequent application of `id` to the concrete parameter, `1`, of type `int`, is transformed to the code shown on line (a). Here, the integer 1 is wrapped, `id` is applied to the wrapped integer, and the result of evaluating the expression is unwrapped and bound to identifier `one`.

A slightly more interesting example is shown in Figure 28. Here, the polymorphic function `first` is defined as one which returns the value found by applying the polymorphic identity function, `id`, to the first of it's parameters.

```
    fun id(x) = x;
    fun first(x,y) = id(x);
    val a = first(1, 2.0);
(a) val a = unwrap(int)(first(wrap(int)(1),wrap(real)(2.0)));
```

Figure 28. Second Example of Polymorphic Function Application in ML.

The application of `first` to the values `1` and `2.0`, the first of type `int` and the second of type `real`, would be compiled as shown on line (a). The two arguments are wrapped using the appropriate instantiation of the `wrap` operator and the function `first` is applied to these wrapped values. Function `first`, in its body, is free to pass the wrapped integer to `id` which, being polymorphic, expects wrapped values itself. Since the result of `first` is of quantified type, it is unwrapped after the return and bound to identifier `a`.

2. Passing a Wrapped Value via a Concrete Formal Parameter.

The remaining instance where conversion between forms is required is when a wrapped object is passed via a concrete formal parameter. As stated earlier, in this case an envelope function is created to perform the proper conversion and apply the monomorphic function to the converted values. The examples of Figures 29 and 30 help clarify the method.

```
fun succ(x:int) = x + 1;

fun apply(f,x) = f(x);

val two = apply(succ, 1);
```

Figure 29. Passing Quantified Actual Parameter via Concrete Formal Parameter.

The problem is set up in Figure 29. Two functions are defined. The first, `succ`, is the monomorphic integer successor function of type `int -> int`; it takes an unwrapped integer as its only parameter. The second function, `apply`, is a polymorphic, higher-order function of type $(\alpha \rightarrow \beta) * \alpha \rightarrow \beta$ that applies it's first parameter, a function of type $\alpha \rightarrow \beta$, to its second parameter, a value of type α, resulting in a value of type β. The function apply is then applied to `succ` and the integer `1`, and the result is bound to identifier `two`.

Since `apply`'s second formal parameter, `x`, is quantified, the second actual parameter (the integer `1`) must be wrapped prior to the call. But this causes difficulties. The function `succ` is monomorphic and takes an unwrapped integer as its parameter; it will not work correctly if it is passed a wrapped integer. For example, if wrapping is performed by boxing the integer, the result is a pointer. The result of applying `succ` to a pointer would be to increment the pointer, causing it to point to whatever was in the next higher word in storage, vice incrementing the integer to which the pointer pointed.

There are several ways to solve this problem (e.g., tagging, passing extra parameters). The method chosen in [Le92] is the use of a program translation to generate an envelope function around the monomorphic function. For the example of Figure 29, the envelope function for `succ` would be as shown in Figure 30(a), with the application of `apply` translated as shown in Figure 30(b).

```
(a) λx.wrap(succ(unwrap(x)))

(b) val two =
      unwrap(apply(λx.wrap(succ(unwrap(x)))), wrap(1));
```

Figure 30. Result of Translating Figure 29.

The envelope function of Figure 30(a) is simply a function abstraction which takes a parameter `x`, unwraps it, applies the monomorphic function `succ` to the unwrapped value, and then wraps the return value.

Figure 30(b) demonstrates the overall translation caused by the application of Figure 29. The envelope function is generated as a local function abstraction and the second actual parameter, the integer `1`, is wrapped. The function `apply` is then applied to these two objects. In the body of `apply`, the envelope function is applied to the wrapped integer. In the body of the envelope function, the integer is unwrapped and the function `succ` is applied to the unwrapped integer. In the body of `succ`, the integer is incremented and the result is returned to the envelope function. The envelope function wraps the integer and returns the wrapped integer to the function `apply`. The function `apply` returns the wrapped integer to the original calling routine where it is unwrapped a final time and bound to the identifier `two`.

It is useful, in the context of wrapping and unwrapping of actual parameters to view `λx.wrap(succ(unwrap(x)))` as the "wrapped" version of `succ`. If the wrapped version is given a name, `succ'`, the translation given in Figure 30 could be expressed more succinctly, as shown in Figure 31.

```
(a) succ' = λx.wrap(succ(unwrap(x)))

(b) val two = unwrap(apply(succ', wrap(1)));
```

Figure 31. Different View of Figure 30.

D. IMPLEMENTATION OF DATA STRUCTURES

So far, we have considered only the implementation of atomic data types. The extension of this scheme to composite data types is straight forward. The following discussion is divided into two parts covering, first, simple composite types (e.g., tuples and records) and, second, the more complex case of recursive data types (e.g., lists).

1. Simple Composite Data Types

Simple composite data types include tuples and records. They are of a fixed, known size and are not recursive. Because of this, their implementation is a simple and direct extension to the implementation already discussed for atomic types.

Passing a record, for example, to a polymorphic function is simply a matter of wrapping each field in the record prior to the call and unwrapping everything after the call. In the case where one of these simple structures is created by the polymorphic function vice being passed to it, the scheme is even simpler: the polymorphic function creates the structure in wrapped form and it is unwrapped upon return. Figure 32 gives an example of this latter case.

In this example, the polymorphic function `mkPair` is defined as one which takes a parameter of quantified type and returns a pair. The call `mkPair(3.14)`, for example, would result in the creation of the pair `(3.14, 3.14)`. Figure 32(a) shows the translation of the call. This translation could be rewritten as shown in Figure 32(b), a notation which might be more comfortable to an imperative language programmer.

Prior to calling `mkPair`, the actual parameter is wrapped. On return, the first and second elements of the tuple are extracted, using the built-in `fst` and `snd` operators, and are unwrapped prior to being bound to the identifier `realPair`.

The clear implication so far is that these simple structures are stored in concrete form. This is often the case but, if the size of the structure is large, wrapping and unwrapping each field can be expensive. In some cases, it might be better to store these

```
    fun mkPair(x) = (x, x);

    val realPair = makePair(3.14);

(a) val realPair = let x = mkPair(wrap(3.14))
                   in (unwrap(fst(x)), unwrap(snd(x)));

(b) val realPair = (unwrap(fst(mkPair(wrap(3.14)))),
                    unwrap(snd(mkPair(wrap(3.14)))));
```

Figure 32. Creation of Data Structure in Polymorphic Function. From [Le92].

structures in a wrapped representation at all times. Figure 33 shows three possibilities, using boxed values as the uniform (wrapped) representation.

Figure 33(a) shows the standard boxed representation used by most current ML implementations. Every field of every record is boxed before being assigned to the record. Here, the record x is represented as a pointer to a set of four boxed values, two of which are reals and two of which are strings. Likewise, record y is a record represented as a pointer to three boxed values, all reals. Access to any of these fields requires unboxing, an inefficient exercise for such routine computations as arithmetic.

A more efficient data representation is seen in Figure 33(b). Here, the real numbers are stored in their unboxed form, while the strings remain boxed, as is typical of most languages. Mixing data types in the manner of record x, however, complicates the object descriptor used by the garbage collector.

A better solution is to re-order the fields of the record so that all unboxed fields are ahead of all boxed fields. The object descriptor then consists of two short integers: one indicating the length of the unboxed part, the other the length of the boxed part. [ShA95]. Figure 33(c) gives an example of this representation.

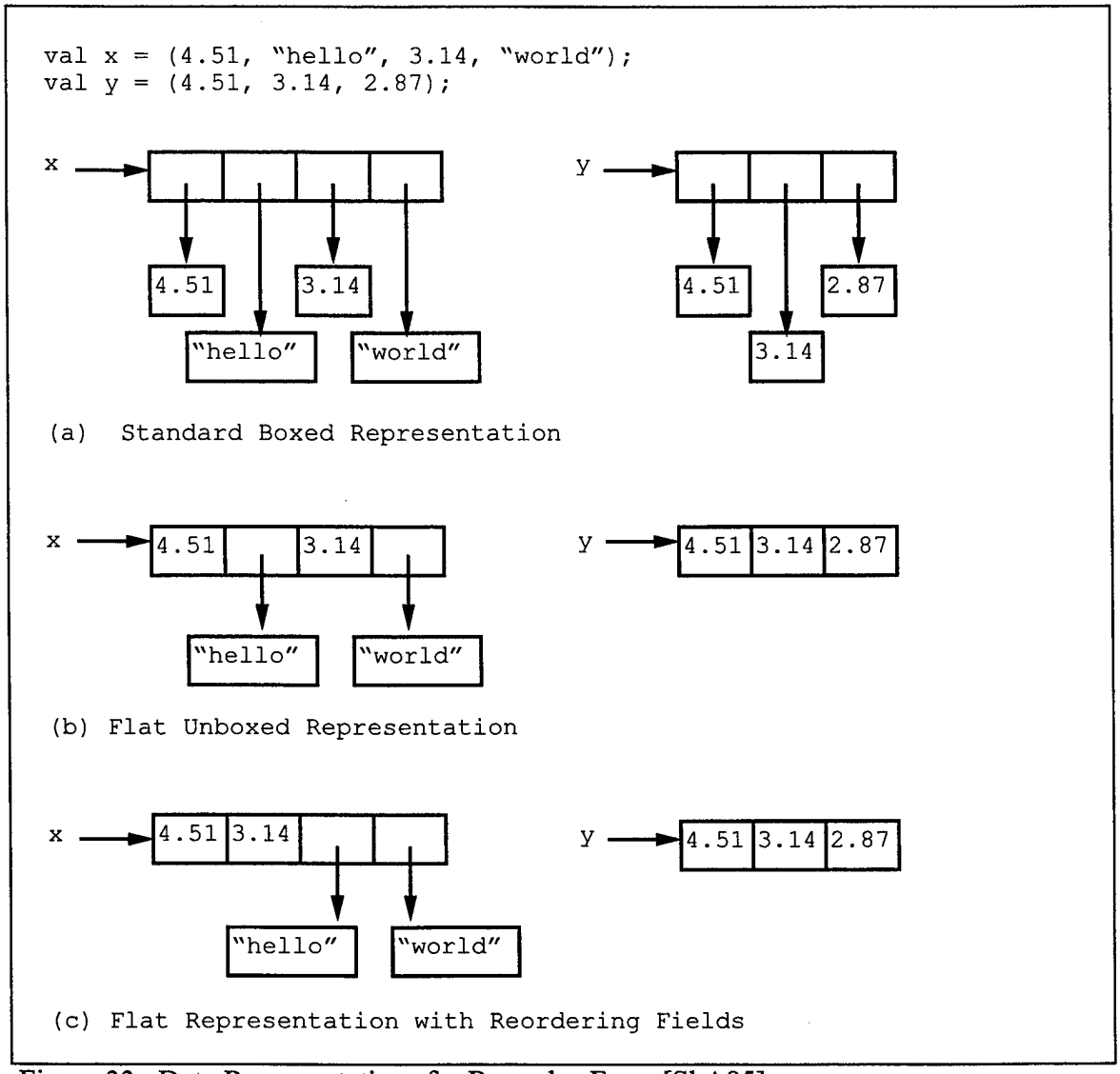

Figure 33. Data Representations for Records. From [ShA95].

This third method is used by both [Le92] and [ShA95], making access to record fields very efficient for monomorphic code. Of course, any unboxed fields will have to be boxed prior to being passed to polymorphic code via quantified formal parameters.

2. Recursive Data Types

Recursive data types are those whose values are composed from values of the same type [Wa90]. One common example suitable for this discussion is the list type in ML, defined as follows:

```
datatype α list = nil | cons of α*α list;
```

In other words, a list is either an empty list or it is a value of type α followed by a list of type α `list`. Any list, then, is built up recursively from the empty list, nil; the list `[1,2,3]` could be written as `cons(1,cons(2,cons(3,nil)))`.

Like all recursive data types, lists are usually represented using pointers. Each element in an α list consists of a record with two fields. The first field is a value of type α; the second field is a pointer to the head of the rest of the list. Thus, the list `L1 = [1,2,3]` could be represented as shown in Figure 34(a).

Polymorphic functions operating on lists, however, are compiled to be independent of the type of the list elements; the polymorphic list reversal function of Figure 4, for example, manipulates individual list elements independent of their type. Subsequently, those elements must be boxed. The standard boxed representation for the list [1,2,3] is shown in Figure 34(b).

But list elements are not always atomic objects. Figure 34(c) shows a flat, unboxed representation of a list composed of three pairs where each element of the list is a pointer to a pair and each pair is represented in the unboxed form discussed in the previous section. This representation is acceptable for functions such as `reverse`, of type α `list -> α list`, but consider the polymorphic function `unzip` [ShA95], of type `(α*β) list -> α list * β list` shown in Figure 35. It operates on lists of type `(α*β) list`, returning a pair of lists, the first of type α list, the second of type β list. The application of `unzip` to the list `[(1,4),(2,5),(3,6)]` results in the pair `([1,2,3],[4,5,6])`.

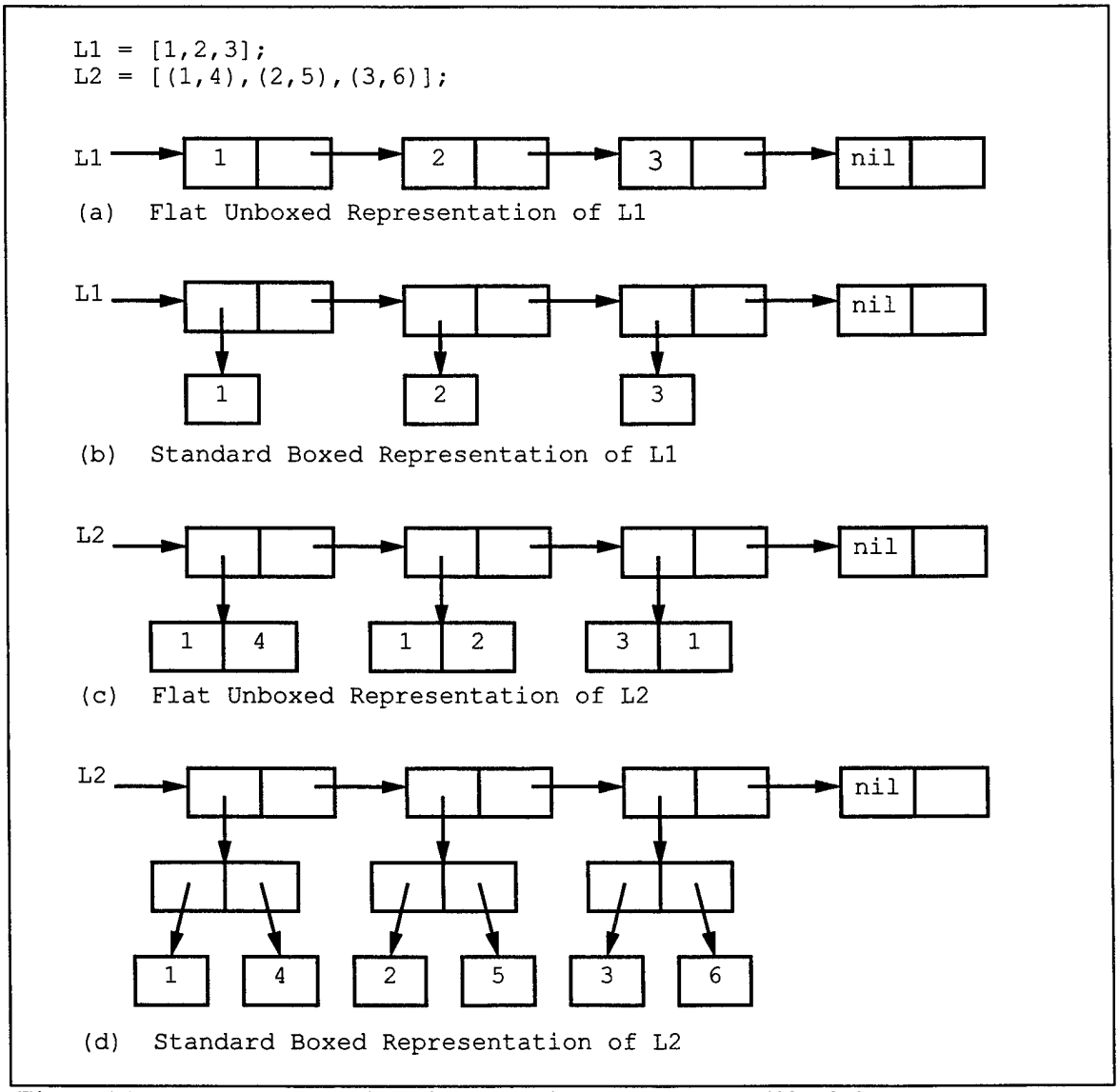

Figure 34. Data Representations for Recursive Types. From [ShA95].

The flat representation of Figure 34(c) is not suitable for function `unzip`, as `unzip` manipulates the individual fields of the pairs comprising the elements of the list. For unzip to work, those fields must be boxed, leading to the standard boxed representation of Figure 34(d). On the same theme, one could imagine having lists of lists of pairs of lists, etc., and could construct polymorphic functions, such as an imaginary `super_unzip`, that require the entire construct to be boxed. Obviously, if

lists were stored in a flat representation, the boxing and unboxing required for such polymorphic function calls could be very expensive.

```
fun unzip l =
  let fun h((a, b)::r,u,w) = h(r,a::u,b::w)
        | h([],u,w) = (reverse u, reverse w)
    in h(l,[],[])
  end
```

Figure 35. Unzip Function in ML. From [ShA95].

For this reason, it is appropriate to store and maintain recursive structures in standard boxed form at all times. Doing so complicates access to, and manipulation of, these structures but it is relatively inexpensive. In the case of lists, for example, appending a new element to the head of a list is simple. The new element is boxed and inserted at the front of the linked list of elements.

Thiemann proposes a revised translation scheme utilizing continuation-passing style and a notion called representation types [Th95]. A result of his work is that many recursive data structures can have more efficient storage representations. These techniques might be applicable to extensions of Polymorphic C.

E. EFFICIENCY AND OPTIMIZATION

The most significant result of the work reported in [Le92] is the successful introduction of mixed data representations to ML via simple program translations and the introduction of wrap and unwrap operators. In experiments which compared the performance of a compiler utilizing mixed representations and the coercion scheme discussed above to that of an identical compiler using the traditional uniform representation, the former was clearly superior in most instances.

The best results were achieved on programs that performed a great deal of integer and floating point arithmetic, involved a significant amount of looping, and/or performed

a significant number of function calls. These results are due to the fact that much of the code in any real program is monomorphic and, hence, benefitted from the ability to directly access data in its optimal representation.

There was no significant difference for programs which performed a great deal of list processing. This, too, makes sense in that the representation of lists is the same in both cases.

The worst results came from programs which utilized a great deal of polymorphism. The reasons for this are also clear. The coercions required for a polymorphic function call can be quite expensive in some instances.

The results reported by [Le92] are obtained by applying the proposed implementation scheme without additional optimizations. Of the many possible optimizations, the following seem to be the most promising.

1. Compile-time Reductions

There are three important cases, all somewhat related, in which static analysis of a program can result in the elimination of a number of unnecessary coercions.

a. Elimination of Trivial Coercions on Data

Any sequence of calls of the form `wrap(unwrap(x))` or `unwrap(wrap(x))` are trivial and can be replaced with `x`. Consider the example of Figure 36, for example.

The rules for passing unwrapped objects via quantified formal parameters were clear: wrap the actual parameter before the call and, if necessary, unwrap the return value after the return. A naive implementation might analyze the code of Figure 36(a) and, based on that rule, mechanically generate the translation of Figure 36(b). It's clear, however, that one unwrap operation and one wrap operation can be saved by eliminating the trivial coercions prior to the second application of `id`. The resulting code is shown in Figure 36(c).

```
(a) fun id(x) = x;
    fun apply(f, x) = f(x);
    val one = id(apply(id, 1));
(b) val one = unwrap(id(wrap(unwrap(apply(id, wrap(1))))));
(c) val one = unwrap(id(apply(id, wrap(1))));
```

Figure 36. Elimination of Trivial Coercions.

b. Use Inline Monomorphic Functions

When a monomorphic function is applied to wrapped values, an envelope function must be inserted around the monomorphic function to unwrap its parameters, call it, and wrap its return value. If the monomorphic code is sufficiently small, inlining that function would save function call overhead.

c. Monomorphic Expansion

If polymorphic functions are consistently replaced by specialized functions for each instance of application on a new type, the program becomes strictly monomorphic. In this case, polymorphic functions would be used similarly to Ada generic functions or C++ templates: they would merely serve as templates for the creation of monomorphic code. The compiler would be free then to generate optimal data representations for all types.

While we have noted that the growth of code could be enormous if many diverse functions are applied to data of many types, there are some advantages to this approach, especially using the implementation scheme discussed in this chapter. First, the code growth in any particular instance may be manageable for a particular program. If it is not, it is possible to monomorphically expand only a portion of the code - perhaps a particular, high-performance set of functions - while leaving the rest unexpanded.

The ability to selectively perform this sort of time/space optimization is a primary strength of this technique. Thiemann provides additional thoughts on this matter [Th95].

2. "Don't Care" Polymorphism

Thiemann notes an extremely simple optimization which he terms "Don't Care" polymorphism [Th95]. It can be fully explained by an extremely simple example. Given a function `first(x,y) = x`, of type $(\alpha*\beta) \rightarrow \alpha$, it does not matter if `y` is wrapped or unwrapped; it is ignored. Since the function does not need to access `y`, it need not be coerced under any circumstance.

3. Proper Use of Tail Recursion

Thiemann notes an optimization somewhat related to the case of eliminating trivial coercions [Th95]. Consider, once again, the function `apply` and the application `apply(succ, 1)`. Using inline notation, this application is translated to:

```
unwrap(apply(λx.wrap(succ(unwrap(x)))), wrap(1));
```

After the application of `succ` to the unwrapped integer, the return value is wrapped by the envelope function and returned to `apply` where it is immediately returned to the calling routine and unwrapped.

Under these circumstances, there is no need for the envelope function to wrap the object prior to the return; the polymorphic function `apply` does nothing with it after the return except return it to the monomorphic routine which, in the end, wants it in unwrapped form. The techniques in [Th95] eliminate this inefficiency.

V. IMPLEMENTATION RECOMMENDATIONS FOR POLYMORPHIC C

This chapter discusses the implementation of polymorphism in Polymorphic C. Section A reviews the primary issues raised in the case studies of Chapters III and IV. Section B presents a recommendation for Polymorphic C. Section C demonstrates how one might achieve the effect of parametric polymorphism in an imperative language by repeating the examples of Chapter IV using C. Section D concludes the chapter with examples of potential translations, a la [Le92], from Polymorphic C to a target language (Polymorphic C augmented with the wrap and unwrap constructs).

A. REVIEW OF IMPLEMENTATION TECHNIQUES

This section briefly reviews some of the implementation decisions covered in previous chapters with the goal of narrowing the scope of choices for Polymorphic C.

In the implementation of parametric polymorphism, polymorphic functions must be compiled to operate on data that is represented in some uniform form. This requirement can be accommodated by representing all data in a system in a uniform form, as is done in ML, but doing so significantly reduces the efficiency of monomorphic code. This has a significant overall impact on a program since monomorphic code usually comprises 80 - 90 % of the total in a typical program.

A better approach is to store and manipulate values in their optimal concrete representations and convert them to a uniform form only when required in order to accommodate polymorphic functions. Regardless of how it is accomplished, this conversion can only occur at one of two points: before the call to a polymorphic function or after the call to a polymorphic function. Napier88 takes the latter approach, [Le92] the former.

1. Conversion After the Call.

This choice is forced because Napier88 supports both type specialization without call and first-class functions. Hence, the compiler is not able to determine statically whether a function being called will be polymorphic or monomorphic at run time. The only efficient alternative is to leave data in concrete form and allow the function, if it is polymorphic, to itself convert the data to uniform form and reconvert it to concrete form prior to return.

However, this conversion scheme demands that the polymorphic function be supplied with the type information it will need in order to perform the appropriate conversions. If the data is of some compound type, it must also be supplied with information regarding the structure of the data. There are three ways to accomplish this. The first is to simply pass this information as separate parameters to the polymorphic function. The second is to tag data with type information.

The third, used in Napier88, is to create an envelope function and store the type and structure information in variables local to that function. The polymorphic code is then dynamically bound to the envelope function, making those local variables visible to the polymorphic function which can exploit that information to perform the correct conversions. On subsequent specializations, the polymorphic function is bound to different envelope procedures with different values in the local variables, resulting in different conversions.

2. Conversion Before the Call

The approach used in Napier88 is not applicable to ML. First, functions in ML are statically bound at time of definition so we are not able to dynamically re-bind polymorphic functions to different envelope functions. Therefore, to convert after the call we must either pass type and structure information via separate parameters or use tagged data.

Secondly, ML does not support type specialization. This means that the compiler is able to statically type the entire program. Hence, the requirement to delay conversion

until after the call to a polymorphic function does not exist; we are free to convert values prior to the call. This is the approach adopted in [Le92]. In this case, the data is converted from within an environment where type and structure information is known; there is no requirement to somehow propagate this information to the polymorphic function.

The only difficulty with this conversion scheme occurs when a polymorphic function must apply a monomorphic function to its data (e.g., a polymorphic sort routine). The monomorphic function expects to receive its data in concrete form but the polymorphic function does not know how to perform the proper conversions. This problem is solved in Leroy's method by creating an envelope function which performs the conversions then applies the monomorphic function to the converted values.

B. APPLICABILITY TO POLYMORPHIC C

Polymorphic C is a strongly and statically typed language. As such, it shares the same implementation constraints as ML. Because Polymorphic C is statically typed, the implementation technique used in Napier88, which relied on dynamic binding, is not applicable to Polymorphic C.

We are free to convert either prior to or after the call to a polymorphic function, with the only concern being that of implementation efficiency. There are several clear alternatives which are outlined in Figure 37.

The most efficient of the alternatives is to convert values to uniform form prior to the call to a polymorphic function. This will lead to the overhead associated with envelope procedures, but only in the case where monomorphic code is applied to data represented in uniform form. In all other cases, the overhead imposed by this scheme is limited to that required to perform the conversions. This method is that given in [Le92].

1. **Conversion After the Call**
   a. ***Pass Type and Structure Information via Additional Parameters***
      Imposes additional function call overhead for every polymorphic function in the system.
   b. ***Use Tagged Data***
      Imposes additional complexity and overhead for the entire system, whether polymorphic or not.

2. **Conversion Before the Call**
   a. ***Create Envelope Functions For Monomorphic Functions***
      Imposes additional function call overhead only when monomorphic functions are applied to uniform values.

Figure 37. Implementation Alternatives for Polymorphic C.

C. SIMULATING PARAMETRIC POLYMORPHISM IN C

Leroy's translation works by automatically inserting appropriate coercions whenever polymorphic functions are specialized. Since C does not support type abstraction (i.e., all types are concrete), one cannot actually apply Leroy's translation to C source code. However, it is possible to achieve the same effect by manually introducing coercions that correspond to those introduced when the translations are applied to equivalent functions in ML.

One first needs to define the uniform representation to be used in C. Since pointers are typed in C, the boxing of a value results in a pointer to a specific type. Hence, we must not only box the objects but also coerce the results to or from a specific pointer type. The choice of type is unimportant as long as it uniform and consistent.

We have arbitrarily chosen the pointer to void (`void*`) as the uniform data representation. Wrapping is performed by referencing an object and casting the result as a pointer to void. Unwrapping is performed by casting a pointer to void to a pointer of the appropriate type and dereferencing the result. An unwrapped integer value `x`, is wrapped by the operations `(void*)&x`. A wrapped integer value `y`, is unwrapped by

the operations `*(int*)y`. These operations may be encapsulated; the `wrap` and `unwrap` functions for types `int` and `float` are shown in Figure 38.

```
void* wrap_int(int &c){return (void*)&c;}
void* wrap_float(float &c){return (void*)&c;}
int unwrap_int(void *u){return *(int*)u;}
float unwrap_float(void *u){return *(float*)u;}
```

Figure 38. Wrap and Unwrap Functions in C.

The discussion continues in two parts. We review some examples of how parametric polymorphism might be achieved in C and conclude with a discussion of how this might be improved.

1. Examples

We repeat here the examples of Chapter IV. The equivalent ML code from previous examples is included as comments immediately preceding the corresponding C code. The `wrap` and `unwrap` functions shown in Figure 38 are contained in the file "wrpunwrp.h".

a. Polymorphic Identity Function

The polymorphic identity function of Figure 27 is coded in C as shown in Figure 39. As can be seen, there is a clear mapping between the ML and C code.

b. Polymorphic Function "first"

The polymorphic function `first` of Figure 28 is coded in C as shown in Figure 40. While still quite simplistic, the call `id(x)` within `first` does serve to demonstrate the passing of a quantified actual parameter via a quantified formal parameter.

```
#include "wrpunwrp.h"

// fun id(x) = x;
void* id(void *x){return x;}

void main()
{
// val one = unwrap(int)(id(wrap(int)(1)));
int one = unwrap_int(id(wrap_int(1)));
}
```

Figure 39. Polymorphic Identity Function.

```
#include "wrpunwrp.h"

// fun id(x) = x;
void* id(void *x){return x;}

// fun first(x, y) = id(x);
void* first(void *x, void *y){return id(x);}

void main()
{
  // val a = unwrap(int)(first(wrap(int)(1),wrap(float)(2.0)));
  int a = unwrap_int(first(wrap_int(1), wrap_float(2.0)));
}
```

Figure 40. Polymorphic Function "first" in C.

c. *Polymorphic Higher-Order Function "Apply"*

The higher-order function apply of Figures 29-31 is coded in C as shown in Figure 41. Note that the use of a function pointer, `*PF`, is required in the call to `apply` because C does not allow higher-order functions.

Also note the function `succ_prime` which represents the wrapped version of `succ`. The use of this function is required because C does not allow nested

function declarations. In other words, there is no C equivalent of the ML expression `apply(λx.wrap(int)(succ(...)), ...)` of Figure 30. Rather, we must mirror the construct of Figure 31, where the anonymous lambda abstraction was given a name, `succ'`, and called explicitly as follows: `apply(succ', ...)`.

```
#include "wrpunwrp.h"

// fun succ(x) = x + 1;
int succ(int x){return x + 1;}

//   succ' = λx.wrap(int)(succ(unwrap(int)(x)))
void* succ_prime(void* x)
{
  return wrap_int(succ(unwrap(x)))
}

typedef void* (*PF)(void*);

// fun apply(f, x) = f(x);
void* apply(PF f, void *x){return f(x);}

void main()
{
  // let two = unwrap_int(apply(succ', wrap(int)(1)))
  int two = unwrap_int(apply(succ_prime, wrap_int(1)));
}
```

Figure 41. Polymorphic Higher-Order Function "Apply" in C.

d. Polymorphic Function "mkPair"

The polymorphic function mkPair of Figure 32 is coded in C as shown in Figure 42. We demonstrate two ways to implement the function application, both of which are also shown in Figure 32.

Since C does not have predefined pairs, we start by declaring structures that mimic the pairs of ML. The structure `floatPair` is the flat unboxed

representation for a pair of floats; `polyPair` is the standard boxed representation for a pair of anything.

In the actual code, there are only two notable differences. The first is the use of a temporary variable, `pp`, in `mkPair`; in C, one cannot have a constructor as the right-hand side of a return statement. The other is the use of a global variable, `x`, in the first call to `mkPair`; in the ML code, the variable is local to the call. Both of these differences are artifacts of C and are not inherently associated with imperative languages.

```
#include "wrpunwrp.h"

struct floatPair {float fst; float snd;};
struct polyPair {void *fst; void *snd;};

// fun mkPair(x) = (x,x);
polyPair mkPair(void *x)
{
  polyPair pp = {x,x};
  return pp;
}

void main()
{
  // val realPair =
  //       let x = mkPair(wrap(float)(3.14))
  //          in (unwrap(float)(fst(x)), unwrap(float)((snd(x)));
  polyPair x = mkPair(wrap_float(3.14));
  floatPair realPair = {unwrap_float(x.fst),
unwrap_float(x.snd)};

  // val realPair =
(unwrap(float)(fst(mkPair(wrap(float)(3.14)))),
  //
unwrap(float)(snd(mkPair(wrap(float)(3.14))));
  floatPair realPair2 =
               {unwrap_float(mkPair(wrap_float(3.14)).fst),
                unwrap_float(mkPair(wrap_float(3.14)).snd)};
}
```

Figure 42. Polymorphic Function "mkPair" in C.

2. Discussion

The previous examples demonstrate that explicit coercions can be used to achieve parametric polymorphism in C. However, there are difficulties, not the least of which is that the programmer must correctly manage the complexity associated with these explicit coercions with little or no help from the compiler. For example, there is nothing in C's type system to prevent the programmer from performing inadvertent casts such as

```
float f = unwrap_float(apply(succ_prime, wrap_int(x)));
```

A second difficulty is that the programmer must explicitly deal with the polymorphic nature of a function to begin with. In the case of `apply`, for example, he/she must insert the proper coercions when `apply` is called and must also explicitly generate the envelope function `succ_prime`. Not only does this decrease programmer productivity and introduce additional sources of error, it also implies some knowledge of the body of `apply` and of the fact that `succ_prime` has not yet been coded by some other programmer. These observations may seem trivial for this toy example but, in the context of a large development effort, it is not clear that a programmer will have this knowledge.

A better approach would be to design an imperative language and a set of appropriate translations such that these coercions and envelope procedures could be automatically generated if and when they are needed. The programmer would then be relieved of the burden associated with simulating polymorphism.

If that language also supported type inferencing, entire programs could be written without any explicit type information at all. Figure 43(a) shows how the C source code of Figure 41 might be improved in an imaginary language which resembles C without explicit type information. With no further assumptions, it would be possible for a compiler to use the method proposed in [Le92] to translate the call to `apply` as shown in Figure 43(b).

This hypothetical language corresponds to Polymorphic C.

```
(a)   succ(x){return x + 1;}

      apply(f, x){return f(x);}

      void main(){two = apply(succ, 1);}

(b)   succ_prime(x){wrap_int(succ(unwrap_int(x)));}
      two = apply(succ_prime, wrap(1));
```

Figure 43. Code Without Explicit Type Information.

D. LEROY'S METHOD APPLIED TO POLYMORPHIC C

One of the great strengths of the method under consideration is that the coercions inserted into function calls are based exclusively on the types of the functions involved. They are completely independent of the body of the function. For example, the call `foo(x)` for any function `foo` of type $\alpha \rightarrow \alpha$, whether `foo` is the simple identity function or one of extreme complexity, is translated to a call of the form: `unwrap(`τ`)(foo(wrap(`τ`))`. Likewise, the call `bar(f, x)` for any function `bar` of type $(\alpha \rightarrow \beta) * \alpha \rightarrow \beta$ is translated to a call of the form:
`unwrap(`τ`)(bar(`λ`y.wrap(`τ`)(f(unwrap(`τ`)(y))),wrap(`τ`)(x)))`

This makes the application to Polymorphic C relatively straightforward. If the target language of the translations is Polymorphic C augmented with `wrap` and `unwrap` operations, the results would be identical to those seen in ML and simulated in C.

Two simple examples will suffice. Figure 44(a) gives the Polymorphic C version of the polymorphic identity function and an application of that function. The resulting translation of the call is shown in Figure 44(b).

```
(a) let id = λx.x
    in
    id(3)

(b) unwrap(id(wrap(3))
```

Figure 44. Polymorphic Identity Function in Polymorphic C.

Figure 45(a) gives the Polymorphic C versions of the integer successor function, `succ`, and the polymorphic, higher-order function `apply`, and an application of `apply` to `succ` and the integer `1`. The translation of the call is shown in Figure 45(b).

```
(a) let succ = λx.x + 1
    in
    let apply = λf,x.f(x)
    in
    apply(succ, 1)

(b) let succ' = λy.wrap(succ(unwrap(y))
    in
    unwrap(apply(succ', wrap(1)))
```

Figure 45. Higher-Order Function "apply" in Polymorphic C.

VI. FURTHER RESEARCH

While the approach presented in [Le92] shows great promise for the implementation of polymorphism in Polymorphic C, there is much work remaining before such an implementation can be fully realized.

A. TRANSLATION RULES

The first step in such an effort must be to formulate a set of translation rules based on the type system of Polymorphic C which can be used to insert the proper coercions into the function calls. Since these coercions are generated based on the type, vice the syntax or structure, of the functions, the translation should as in [Le92].

Polymorphic C does include two data types not seen in the core ML considered in [Le92]. It includes pointers of type $\tau.ptr$, and arrays. The introduction of pointers should not present great difficulty because they would not have to be coerced. The values attained by dereferencing pointers might have to be coerced but this, too, should be amenable to translation from the typing rules. Likewise, arrays can be handled easily if maintained in standard boxed representation at all times.

B. DATA REPRESENTATIONS

A decision will have to be made concerning the proper data representation for the various data types. At present, the only concrete atomic data types in Polymorphic C are integers and pointers to integers. The only data structures are arrays. Any implementation decisions made now regarding data representations should anticipate the eventual introduction of additional atomic types such as reals and characters and of data structures such as structures and lists. Thiemann presents some techniques which might be applicable to future extensions to Polymorphic C [Th95].

C. TYPELESS RUN TIME SYSTEM

Given that Polymorphic C is strongly and statically typed, there can be no type insecurities at run time for a properly compiled program. It may be possible to devise a run time system which utilizes this assurance to improve the efficiency of the implementation. Pederson has investigated parameter passing methods using direct manipulation of the run time stack to preclude unnecessary coercions [Pe95]. His results would be applicable to a large number of language implementations.

APPENDIX

This appendix provides a somewhat detailed treatment of the implementation method presented in [Le92], which allows ML to be compiled with mixed data representations. The discussion borrows heavily from that work. Section A provides a formal description of the system. Section B provides translations for the examples shown in Chapter IV.

A. FORMALIZATION

The method presented in [Le92] consists of a translation from a source language to a target language. The source language is core ML. The target language is core ML augmented with the two constructs wrap(τ) and unwrap(τ). The syntax is shown in Figure A1, where x is an identifier, i is an integer constant, f is a floating point constant and α is a type variable.

Source terms: $a ::= i \mid f \mid x \mid \lambda x.a \mid \text{let } x = a_1 \text{ in } a_2 \mid a_1(a_2) \mid (a_1, a_2) \mid \text{fst}(a) \mid \text{snd}(a)$

Target terms: $a' ::= i \mid f \mid x \mid \lambda x.a \mid \text{let } x = a_1 \text{ in } a_2 \mid a_1(a_2) \mid (a_1, a_2) \mid \text{fst}(a) \mid \text{snd}(a)$

$\mid \text{wrap}(\tau)(a') \mid \text{unwrap}(\tau)(a')$

Type expressions: $\tau ::= \alpha \mid \text{int} \mid \text{float} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2$

Type schemes: $\sigma ::= \forall \alpha_1 \ldots \alpha_n . \tau$

Figure A1. Target Language Syntax. From [Le92].

Type inferencing is performed by applying Milner's type discipline to the source language. While we will not concern ourselves here with type inferencing, the typing

rules are important due to the similarity with translation rules introduced next. The typing rules are shown in Figure A2. The predicate $E \mathsf{a}\ x:t$ is read as "under assumption E, term a has type τ". The construct $\frac{A \quad B}{C}$ is read "(A and B) implies C".

[1] $$\frac{E(x) = \forall \alpha_1 ... \alpha_n . \tau \quad Dom(\rho) \subseteq \{\alpha_1 ... \alpha_n\}}{E \mathsf{a}\ x : \rho(\tau)}$$

[2] $E \mapsto i : \text{int}$ [3] $E \mapsto f : \text{float}$

[4] $$\frac{E + x : \tau_1 \mapsto a : \tau_2}{E \mapsto \lambda\, x.a : \tau_1 \to \tau_2}$$

[5] $$\frac{E \mapsto a_2 : \tau_1 \to \tau_2 \quad E \mapsto a_1 : \tau_2}{E \mapsto a_2(a_1) : \tau_2}$$

[6] $$\frac{E \mapsto a_1 ; \tau_1 \quad E \mapsto a_2 : \tau_2}{E \mapsto (a_1, a_2) : \tau_1 \times \tau_2}$$

[7] $$\frac{E \mapsto a : \tau_1 \times \tau_2}{E \mapsto \text{fst}(a) : \tau_1}$$ [8] $$\frac{E \mapsto a : \tau_1 \times \tau_2}{E \mapsto \text{snd}(a) : \tau_2}$$

[9] $$\frac{E \mapsto a_1 : \tau_1 \quad E + x : Gen(\tau_1, E) \mapsto a_2 : \tau_2}{E \mapsto let\ x = a_1\ in\ a_2 : \tau_2}$$

Figure A2. Typing Rules for Core ML. From [Le92].

Once the type system has established a typing for a term and it's sub-terms, the translation rules of Figure A3 are applied. The translation is presented as the predicate $E \mapsto x : t \Rightarrow a'$, which is read as "under assumption E, term a has type t and is translated to the term a'".

[T1] $$\frac{E(x) = \forall \alpha_1 ... \alpha_n . \tau \quad Dom(\rho) \subseteq \{\alpha_1 ... \alpha_n\}}{E \mapsto x : \rho(\tau) \Rightarrow S_\rho(x : \tau)}$$

[T2] $E \mapsto i : \text{int} \Rightarrow i$ [T3] $E \mapsto f : \text{float} \Rightarrow f$

[T4] $$\frac{E + x : \tau_1 \mapsto a : \tau_2 \Rightarrow a'}{E \mapsto \lambda x.a : \tau_1 \to \tau_2 \Rightarrow \lambda x.a'}$$

[T5] $$\frac{E \mapsto a_2 : \tau_1 \to \tau_2 \Rightarrow a'_2 \quad E \mapsto a_1 : \tau_2 \Rightarrow a'_1}{E \mapsto a_2(a_1) : \tau_2 \Rightarrow a'_2(a'_1)}$$

[T6] $$\frac{E \mapsto a_1 ; \tau_1 \Rightarrow a'_1 \quad E \mapsto a_2 : \tau_2 \Rightarrow a'_2}{E \mapsto (a_1, a_2) : \tau_1 \times \tau_2 \Rightarrow (a'_1, a'_2)}$$

[T7] $$\frac{E \mapsto a : \tau_1 \times \tau_2 \Rightarrow a'}{E \mapsto \text{fst}(a) : \tau_1 \Rightarrow \text{fst}(a')}$$ [T8] $$\frac{E \mapsto a : \tau_1 \times \tau_2 \Rightarrow a'}{E \mapsto \text{snd}(a) : \tau_2 \Rightarrow \text{snd}(a')}$$

[T9] $$\frac{E \mapsto a_1 : \tau_1 \Rightarrow a'_1 \quad E + x : Gen(\tau_1, E) \mapsto a_2 : \tau_2 \Rightarrow a'_2}{E \mapsto let\ x = a_1\ in\ a_2 : \tau_2 \Rightarrow let\ a'_1\ in\ a'_2}$$

Figure A3. Translation Rules. From [Le92].

The heart of the translation is seen in the rule [T1] which is the rule for type specialization. E is the environment, a mapping from identifiers to type schemes and ρ is a of types for type variables. Loosely translated, the rule [T1] says: if there exists in E a mapping from some identifier x to some type scheme $\forall\alpha_1...\alpha_n$ and the types $\alpha_1...\alpha_n$ are type variables in the domain of ρ, then x has a type defined by replacing each occurrence of the type variable τ with the type to which τ is mapped in ρ and is translated by applying the transformation S_ρ.

We shall return to the S transformation. For now, consider the concrete example of Figure A4 which shows the process by which a polymorphic, higher-order function is specialized. Initially, E and ρ are both empty. When the integer successor function,

succ, is defined, identifier succ is added to the domain of *E* and is associated with the type int -> int. When the function apply is defined, the process is repeated for identifier apply and the type (α -> β) * α -> β. Variable anInt is handled likewise.

When apply is applied to succ and anInt, the type system is able to infer that both α and β are both of type integer for this application; this mapping from type variables to types for a given application is what ρ represents. As a result, apply is assigned the type (int -> int) * int -> int for this application.

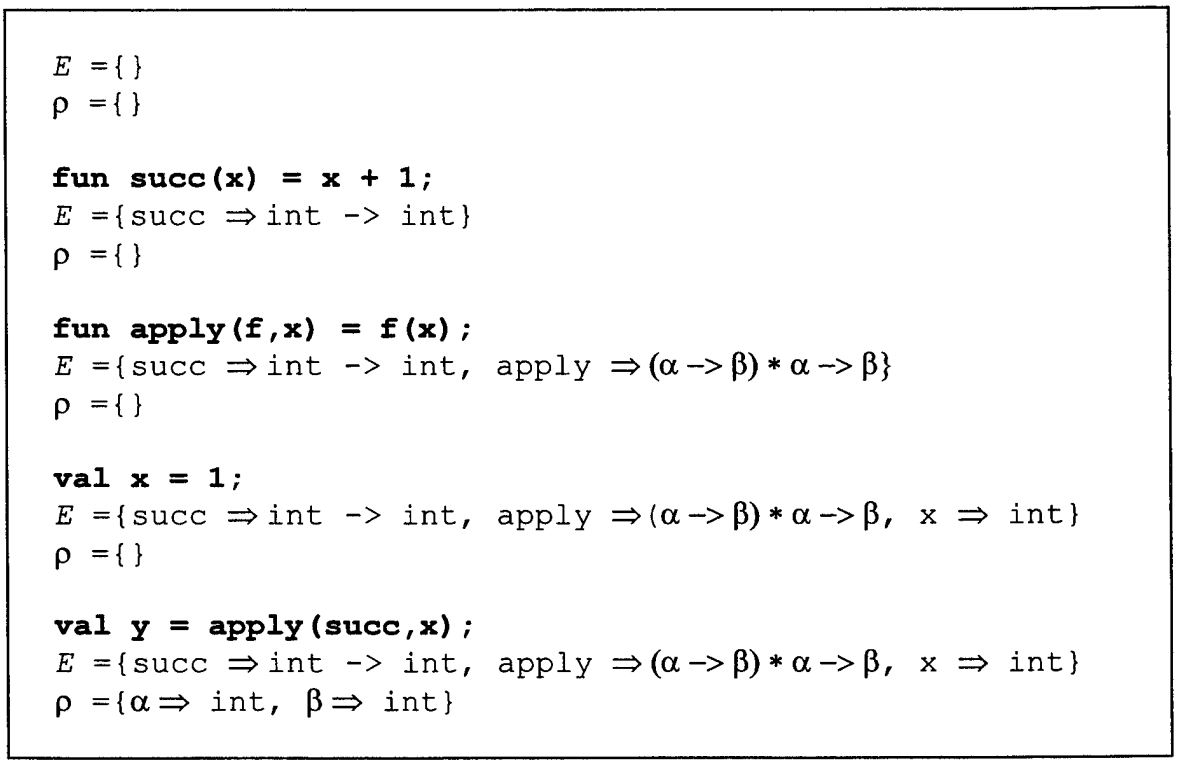

Figure A4. Mappings of *E* and ρ.

The work of the translation is performed by the transformations *S* and *G*, shown in Figures A5 and A6. In our example, translation rule [T1] invokes the translation $S_\rho(x{:}\tau)$, where x is the function apply and τ is the type (int->int)*int->int.
In the notation of the translation rules, a' = x. We first apply *S* transformation rule [S5], where a' = apply, t_1 = (int->int)*int, and t_2 = int, resulting in the term

λx.Sρ(apply(Gρ(x:(int->int)*int)):int). The rest of the transformation is straight forward and is shown in Figure A9.

[S1] $S\rho(a' : \alpha) = \text{unwrap}(\rho(\alpha))(a')$

[S2] $S\rho(a' : int) = a'$

[S3] $S\rho(a' : float) = a'$

[S4] $S\rho(a' : \tau_1 \times \tau_2) = \text{let } x = a' \text{ in } (S\rho(\text{fst}(x) : \tau_1), S\rho(\text{snd}(x) : \tau_2))$

[S5] $S\rho(a' : \tau_1 \rightarrow \tau_2) = \lambda x.S\rho(a'(G\rho(x : \tau_1)) : \tau_2)$, where x is not free in a'

Figure A5. S Transformations. From [Le92].

[G1] $G\rho(a' : \alpha) = \text{wrap}(\rho(\alpha))(a')$

[G2] $G\rho(a' : int) = a'$

[G3] $G\rho(a' : float) = a'$

[G4] $G\rho(a' : \tau_1 \times \tau_2) = \text{let } x = a' \text{ in } (G\rho(\text{fst}(x) : \tau_1), G\rho(\text{snd}(x) : \tau_2))$

[G5] $G\rho(a' : \tau_1 \rightarrow \tau_2) = \lambda x.S\rho(a'(S\rho(x : \tau_1)) : \tau_2)$, where x is not free in a'

Figure A6. G Transformations. [From Le92].

B. TRANSLATIONS

The following translations are provided for examples given in Chapter IV. In Figures A7 through A10, part (a) displays the source code, part (b) displays the function application which invokes the translation, part (c) shows the translation as derived from the translation and transformation rules of Section A, and part (d) shows the final results.

(a) `fun id(x) = x;`

(b) `val one = id(1);`

(c) $$\frac{E \mapsto \mathrm{id} : \mathrm{int} \to \mathrm{int} \Rightarrow S\rho(id : \alpha \to \alpha) \qquad E \mapsto 1 : \mathrm{int} \Rightarrow 1}{E \mapsto \mathrm{id}(1) : \mathrm{int} \Rightarrow S\rho(\mathrm{id} : \alpha \to \alpha)(1)}$$ [T5], [T1], [T3]

$$S\rho(\mathrm{id} : \alpha \text{ -> } \alpha) = \lambda x.S\rho(\mathrm{id}(G\rho(x : \mathrm{int})) : \mathrm{int})$$ [S5]

$$= \lambda x.S\rho(\mathrm{id}(\mathrm{wrap}(\mathrm{int})(x)) : \mathrm{int})$$ [G1]

$$= \lambda x.\mathrm{unwrap}(\mathrm{int})(\mathrm{id}(\mathrm{wrap}(\mathrm{int})(x)))$$ [S1]

(d) $\mathrm{id}(1) = \mathrm{unwrap}(\mathrm{int})(\mathrm{id}(\mathrm{wrap}(\mathrm{int})(1)))$

Figure A7. Translation of Polymorphic Identity Function (see Figure 31).

(a)
```
fun id(x) = x;
fun first(x,y) = id(x);
```

(b)
```
val a = first(1, 2.0);
```

$E \mapsto \text{first} : \text{int} \times \text{float} \to \text{int} \Rightarrow S\rho(\text{first} : \alpha \times \beta \to \alpha)$

$E \mapsto 1 : \text{int} \Rightarrow 1$

(c)
$$\frac{E \mapsto 2.0 : \text{float} \Rightarrow 2.0}{E \mapsto \text{first}(1, 2.0) : \text{int} \Rightarrow S\rho(\text{first} : \alpha \times \beta \to \alpha)(1, 2.0)} \qquad \text{[T5], [T1], [T2], [T3]}$$

$S\rho(\text{first} : \alpha \text{ x } \beta \text{ -> } \alpha) = \lambda x.S\rho(\text{first}(G\rho(x : \text{int x float})) : \text{int})$ [S5]

$= \lambda x.S\rho(\text{first}(\text{let y = x in } (G\rho(\text{fst(y)} : \text{int}), G\rho(\text{snd(y)} : \text{float}))) : \text{int})$ [G4]

$= \lambda x.S\rho(\text{first}(\text{let y = x in } (\text{wrap(int)(fst(y))}, \text{wrap(float)(snd(y))})) : \text{int})$ [G2], [G3]

$= \lambda x.\text{unwrap(int)}(\text{first}(\text{let y = x in } (\text{wrap(int)(fst(y))}, \text{wrap(float)(snd(y))}))$ [S1]

(d)

first(1,2.0) = unwrap(int)(first(let y = (1,2.0) in (wrap(int)(fst(y)), wrap(float)(snd(y)))))

or, equivalently:

first(1,2.0) = unwrap(int)(first(wrap(int)(1), wrap(float)(2.0)))

Figure A8. Translation of Polymorphic Function First (see Figure 32).

(a)
```
fun succ(x) = x + 1;
fun apply(f, x) = f(x);
```

(b)
```
val two = apply(succ, 1);
```

(c)

$$E \mapsto \text{apply} : (\text{int} \to \text{int}) \times \text{int} \to \text{int} \Rightarrow S\rho(\text{apply} : (\alpha \to \beta) \times \alpha \to \beta)$$

$$E \mapsto \text{succ} : \text{int} \to \text{int} \Rightarrow \text{succ}$$

$$\frac{E \mapsto 1 : \text{int} \Rightarrow 1}{E \mapsto \text{apply}(\text{succ}, 1) : \text{int} \Rightarrow s\rho(\text{apply} : (\alpha \to \beta) \times \alpha \to \beta)(\text{succ}, 1)} \quad \text{[T5], [T1], [T2]}$$

$S\rho$(apply : (α -> β) x α -> β) = λx.Sρ(apply($G\rho$(x: (int -> int) x int)): int) [S5]

= λx.Sρ(apply(let y = x in (Gρ(fst(y): int -> int), $G\rho$(snd(y): int))): int) [G4]

= λx.Sρ(apply(let y = x in (Gρ(fst(y): int -> int), wrap(int)(snd(y)))): int) [G1]

= λx.Sρ(apply(let y = x in (λz.$G\rho$(fst(y)($S\rho$(z: int)): int), wrap(int)(snd(y)))): int) [G5]

= λx.Sρ(apply(let y = x in (λz.$G\rho$(fst(y)(unwrap(int)(z)): int), wrap(int)(snd(y)))): int) [S1]

= λx.Sρ(apply(let y = x in (λz.wrap(int)(fst(y)(unwrap(int)(z))), wrap(int)(snd(y)))): int) [G1]

= λx.unwrap(int)(apply(let y=x in (λz.wrap(int)(fst(y)(unwrap(int)(z))), wrap(int)(snd(y))))) [S1]

(d)

apply(succ, 1) = unwrap(int)(apply(let y = (succ, 1) in (λz.wrap(int)(fst(y)(unwrap(int)(z))), wrap(int)(snd(y)))))

= unwrap(int)(apply(λz.wrap(int)(fst(succ, 1)(unwrap(int)(z))), wrap(int)(snd(succ, 1))))

= unwrap(int)(apply(λz.wrap(int)(succ(unwrap(int)(z))), wrap(int)(1)))

= unwrap(int)(apply(succ', wrap(int)(1))), where succ' = λz.wrap(int)(succ(unwrap(int)(z)))

Figure A9. Translation of Higher-Order Function Apply (see Figures 33 - 35).

(a) `fun mkPair(x) = (x, x);`

(b) `val realPair = mkPair(3.14);`

(c)
$$\frac{E \mapsto \text{mkPair} : \text{float} \to \text{float} \times \text{float} \Rightarrow S\rho(\text{mkPair} : \alpha \to \alpha \times \alpha) \quad E \mapsto 3.14 : \text{float} \Rightarrow 3.14}{E \mapsto \text{mkPair}(3.14) : \text{float} \Rightarrow s\rho(\text{mkPair} : (\alpha \to \alpha \times \alpha)(3.14)} \quad \text{[T5], [T1], [T3]}$$

$S\rho$(mkPair : α -> α x α) = λx.Sρ(mkPair($G\rho$(x: float)): float * float) [S5]

= λx.Sρ(mkPair(wrap(float)(x)): float x float) [G1]

= λx.let y = mkPair(wrap(float)(x)) in (Sρ(fst(y): float), Sρ(snd(y): float)) [S4]

= λx.let y = mkPair(wrap(float)(x)) in (unwrap(float)(fst(y)), unwrap(float)(snd(y))) [S1]

(d)

mkPair(3.14) = let y = mkPair(wrap(float)(3.14)) in (unwrap(float)(fst(y)), unwrap(float)(snd(y)))

or, equivalently

mkPair(3.14) = (unwrap(float)(fst(mkPair(wrap(float)(3.14)))),
unwrap(float)(snd(mkPair(wrap(float)(3.14)))))

Figure A10. Translation of Polymorphic Function mkPair (see Figure 37).

LIST OF REFERENCES

[Car84] Cardelli, L., Basic Polymorphic Type Checking, *AT&T Bell Laboratories Computing Science Technical Report 119*, 1984.

[CaW85] Cardelli, L. and Wegner, P., On Understanding Types, Data Abstraction and Polymorphism, *Computing Surveys*, 17/4, 1985.

[CHJ94] Carlson, W., Hudak, P., Jones, M., An Experiment Using Haskell to Prototype "Geometric Region Servers" for Navy Command and Control, *Yale University Research Report 1031*, 1994.

[HaL94] Harper, R. and Lee, P., Advanced Languages for Systems Software: The Fox Project in 1994, *Carnegie-Mellon University Technical Report 94-105*, 1994.

[HuJ94] Hudak, P. and Jones, M., Haskell vs. Ada vs. Awk vs. ...: An experiment in Software Prototyping Productivity, Yale University, 1994.

[KR78] Kernighan, B. and Richie, D., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Le92] Leroy, X., Unboxed Objects and Polymorphic Typing, *Proc. 19th ACM Symposium on Principles of Programming Languages*, pp. 177-188, 1992.

[MDCB91] Morrison, R., Dearle, A., Connor, R. C. H. and Brown, A. L., An Ad Hoc Approach to the Implementation of Polymorphism, *ACM Transactions on Programming Languages and Systems*, 13/3, 1991.

[Pe95] Pederson, C., Uniform Representation in Polymorphic C, NPSCP-95-004, September 1995.

[PJ91] Peyton-Jones, S. L., Unboxed Values as First-Class Citizens in a Non-Strict Functional Language, *Proc. 5th ACM Conf. on Functional Programming and Computer Architecture, LNCS* 523, Springer-Verlag, pp. 637-666.

[ShA95] Shao, Z. and Appel, A., A Type-Based Compiler for Standard ML, *Proc. 1995 Conf. on Programming Language Design and Implementation*, pp 116-129, 1995.

[SmVo95] Smith, G. and Volpano, D., An ML-style Polymorphic Type System for C, Unpublished Manuscript.

[St92] Stansifer, R., *ML Primer*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Str91] Stroustrup, B., *The C++ Programming Language, 2nd Edition*, Addison-Wesley, Reading, MA, 1991.

[Th95] Thiemann, P., Unboxed Values and Polymorphic Typing Revisited, *Record of ACM Conf. on Functional Programming and Computer Architecture*, pp. 24-35, 1995.

[Wa90] Watt, D., *Programming Language Concepts and Paradigms*, Prentice-Hall Intl, Hertfordshire, UK, 1990.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   Cameron Station
   Alexandria, Virginia 22304-6145

2. Dudley Knox Library 2
   Code 013
   Naval Postgraduate School
   Monterey, California 93943-5101

3. Chairman, Code CS 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943-5101

4. Dr. Dennis Volpano, Code CS/VO 2
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943-5101

5. Dr. Thomas Wu, Code CS/WQ 1
   Department of Computer Science
   Naval Postgraduate School
   Monterey, California 93943-5101

6. Dr. Geoffrey Smith 1
   School of Computer Science
   Florida International University
   University Park
   Miami, Florida 33199

7. Peter B. Bonem 2
   c/o Donald S. Bonem
   6718 South 150th Street
   Omaha, Nebraska 68137